HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

DISSERTATION

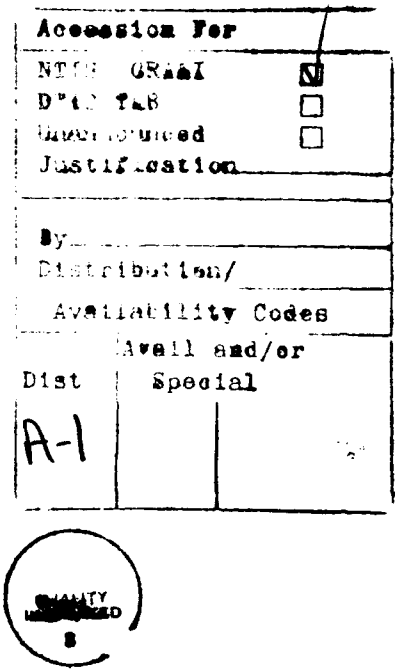
Michael Alan Dukes
Captain, US Army

AFIT/DS/ENG/92-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/92-1



HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

DISSERTATION

Michael Alan Dukes
Captain, US Army

AFIT/DS/ENG/92-1

Approved for public release; distribution unlimited

AFIT/DS/ENG/92-1

HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Michael Alan Dukes, B.S., M.S.E.E

Captain, US Army

March, 1992

AFIT/DS/ENG/92-1

HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

Michael Alan Dukes, B.S., M.S.E.E.

Captain, USA

Approved:

Frank M. Brown, Chairman	3 MARCH 92
Joanne E. DeGroat (The Ohio State University)	5 March 92
Matthew Kabrisky	3 MAR 92
Mark A. Mehalic	4 Mar 92
Henry Potoczny	3 MARCH 1992

Accepted:

J. S. Przemieniecki
Institute Senior Dean

Preface

When I first began work on logic extraction it appeared that the process of deriving hierarchy from a "sea" of transistors was intuitive. From observing engineers working on layout descriptions, it appeared that the human's ability to generate simple gate-level schematics from a transistor netlist was as simple as a bird taking to flight. However, as the process of a bird flying is more complicated than a matter of flapping its wings, logic extraction is more complicated than a matter of just matching a template or a rule. There are properties of the bird not readily apparent to the observer, such as the aerodynamics of the wings and the coordination of guidance between the tail and the wings. In much the same manner, there are properties of logic extraction not readily apparent to the engineer, such as connectivity, internal and external, of an extracted component.

Just as the Wright brothers captured the essence of flight, I have captured the essence of logic extraction. For the past several years, others have tried to perform logic extraction and thought that they had achieved it. Now that logic extraction is presented in a formal fashion in this dissertation, I hope that the interested individual will see that it is not as simple as it appears. I also hope that using logic extraction for hardware-verification will become more popular than simulation.

I wish to acknowledge my committee for the time they invested proof-reading my dissertation. My committee members, Dr. Frank M. Brown, Dr. Joanne E. DeGroat, Dr. Matthew Kabrisky, Dr. Mark A. Mehalic, and Dr. Henry Potoczny, all provided helpful comments and support in my research. I am especially grateful to Dr. Frank M. Brown for keeping all the details in the right order and to Dr. Joanne E. DeGroat for keeping the macroview of the research in perspective. The talents of both provided the appropriate balance.

I also wish to express my appreciation to the Solid State Electronics Directorate of Wright Laboratory. Wright Laboratory provided all of the equipment and lab space for my work. Working at Wright Laboratory helped provide additional insight into the needs of the Army and Air Force,

which helped guide my work. I am indebted to Dr. John Hines who orchestrated all of the support and Darrell Barker for the time he spent ensuring all of the equipment was arriving when needed. I also want to mention the support I received from Luis Concha and Captain Karen Serafino who performed related work to my research as well as Debora McDivitt and Valerie Holler who ensured I arrived at various TDY locations when needed. If I have left anyone out, I apologize, it was not intentional.

The dissertation contains nine chapters and four appendices. Chapter 1 is an introduction to the dissertation. Chapter 2 discusses different methods used in comparing behavioral specifications, structural specifications, and layout specifications. Chapter 3 contains a survey of past attempts at logic extraction and a description of the structural specification and layout specification used in the dissertation. Chapter 4 demonstrates the consistency, completeness, and termination properties of logic extraction. In Chapter 5, several hardware delay models used to demonstrate the feasibility of pin-to-pin critical path analysis are presented. A pin-to-pin critical path analysis procedure with logic extraction is discussed in Chapter 6. Some results using logic extraction are presented in Chapter 7. Chapter 8 lists some limitations that impact on the completeness of logic extraction. Chapter 9 lists conclusions and recommendations for future work.

Several appendices are provided with the dissertation. They serve to provide some background material and support to the content of the dissertation. Appendix A contains definitions of terms and concepts used in the dissertation. Appendix B contains suggestions for improving the efficiency of the logic extraction process. Appendix C is a demonstration of how HOL is used to compare a behavioral specification and a structural specification. Appendix D is an approach to translating VHDL data-flow models to VHDL structural models. Finally, Appendix E is a brief discussion of formal methods.

Michael Alan Dukes

Table of Contents

List of Acronyms

Acronym	Explanation
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BIST	Built-In Self Test
CMOS	Complementary Metal-Oxide-Semiconductor
CAD	Computer Aided Design
cif	Caltech Intermediate Format
DOD	Department of Defense
DRC	Design-Rule Check
esim	Switch level simulator
GES	Generalized Extraction System
GND	Ground or Zero Volts
HDL	Hardware Description Language
HOL	Higher Order Logic
mextra	Translate mask level format to transistor netlist format
mossim	MOS simulator
sim	Transistor netlist generated by mextra
STOVE	Sim To VHDL Extraction
Vdd	Supplied Voltage, High, or +5 Volts
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language

List of Figures

List of Tables

AFIT/DS/ENG/92-1

Abstract

A Prolog-based system is described which employs logic-extraction to perform hardware-verification. The extraction rules are built automatically from hierarchical structural VHDL models, enabling the equivalence of a structural VHDL description and a layout specification to be verified. Pin-to-pin critical-path analysis is performed within the logic-extraction process; many noncritical paths are pruned early, making pin-to-pin critical path analysis of large circuits feasible. It is demonstrated that a design methodology based on logic extraction, VHDL, and a layout tool can provide a fabricated functionally-correct IC design without circuit-level or switch-level simulation. This methodology is shown to be practical for VLSI designs up to 250,000 transistors in size. The properties of correctness, completeness, and guaranteed termination are examined for the extraction process.

HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

I. Introduction

The Department of Defense has adopted VHDL[1] as a standard means of documenting digital designs. A structural description in VHDL is an orderly top-down hierarchical decomposition of a circuit into sub-structures; these are comprehensible, at every level, to the designer. The ultimate product of design, however, is a transistor-layout, a file describing large numbers (up to hundreds of thousands) of interconnected transistors. This file is used to manufacture the circuit in silicon. To be certain that the design complies with its documentation, the designer must somehow convince himself that the layout—representing a seemingly-formless mass of transistors and wires—is a realization of its tidy VHDL description. To do so by inspection is beyond human capability. Instead, the designer today must revert, in software, to the breadboard-testing of earlier days: he checks his design by conducting an input-output experiment, applying a sequence of inputs to a simulated transistor-layout and comparing the resulting outputs with those that would be produced by the VHDL description.

Though greatly assisted by VHDL, stimulus-response experiments are inherently deficient as a way to ensure the compliance of a design with its documentation. The number of required test-inputs increases astronomically as the size of a circuit increases; moreover, memory-elements (present in most circuits) vastly complicate the task of producing and interpreting a test-sequence. Designers continue testing-by-simulation because no practical alternative has been available. The work presented in this dissertation provides one alternative.

[1] VHDL is an acronym for VHSIC Hardware Description Language and VHSIC is an acronym for Very High Speed Integrated Circuit. VHSIC-class integrated circuits include designs larger than 100,000 transistors. VHDL is IEEE Standard 1076-1987 (IEEE 87).

This research is based on the discovery that circuit-extraction, a well-known technique, may be used for purposes which apparently have never before been contemplated. To "extract" a circuit means to begin with a low-level description of its structure and to derive a higher-level description. A typical extraction-system might accept a description of a circuit as an interconnection of transistors and generate a description of the same circuit as an interconnection of components such as registers, adders, and multiplexers.

An original objective was to develop an extraction-system that would accept a transistor-level (or gate-level) description of a circuit and would generate a hierarchical description of the circuit in VHDL. Such extraction has not, to our knowledge, been possible until now, and would be of significant value to the digital-design community. It has turned out to be a relatively simple by-product, however, of the system that has emerged. Called GES (Generalized Extraction System), the system performs the following tasks:

1. **Formal Hardware-Verification.** GES verifies that a hardware design is fully compliant with its hardware documentation. Supplied with a structural description in hierarchical VHDL, GES first produces a custom extractor capable of extracting only the specified structure. GES then attempts an extraction. If the attempt succeeds, the design is verified to be 100% compliant with the documentation. If the attempt fails, then the design may deviate from its documentation in some respect. In the latter case, GES provides diagnostic information which enables the designer quickly to determine the nature and location of the deviation.

2. **Reverse-Engineering of Undocumented Designs.** The DOD has a serious problem in replacing parts whose functional documentation is either incomplete or non-existent. Given a low-level description (at the transistor or gate level), GES will produce a functional description of the circuit. GES also produces timing information and high-level VHDL documentation. The "views" of the circuit that are produced may be tailored to individual requirements.

3. **Detection and Location of Errors in Design.** At different levels in the process of extraction, design-rule checks are performed by GES to identify improperly configured components.

4. **Assistance in Incremental Documented Design.** GES enables documentation and layout to stay in step. At each stage of design, a circuit is guaranteed to comply with its VHDL description; at no point is simulation necessary. The direct use of the VHDL documentation to verify a layout not only encourages a designer to keep his documentation current, it *requires* him to do so. Documentation is typically something conjured *ex post facto*; using GES, however, the documentation becomes an essential part of the process of design. If modifications to a circuit-layout are required, the designer using GES must modify the VHDL documentation first. GES thus provides a useful stimulus to keep documentation current.

5. **Critical-Path Analysis.** GES locates pin-to-pin critical paths in a layout. The requisite timing calculations are based on distributed resistance and capacitance values derived from the layout-description.

The process of formal hardware-verification presented in this paper combines two techniques. The first technique, somewhat similar to that of (Papas 88), uses a rule-based logical extraction process to prove 100% functional compliance between a structural hardware description and its associated component netlist. For true verification of digital hardware, both the functional and temporal aspect of the design must be examined. Thus, a second technique, involving list processing, is used to extract pin-to-pin critical paths from the structural hardware description and its associated component netlist. The critical paths of the hardware model and the component netlist may be compared to ensure that the timing in the circuit meets the restrictions on delays specified in the digital hardware description.

Computer Aided Design of Hardware

The process of CAD development of hardware involves several steps with various tools to aid in the design process. Figure 1 is a diagram of the general flow of the design process as it has existed, void of formal hardware-verification. This process begins with the development of a hardware behavioral specification[2]. Several iterations through simulation may be required to examine the behavior and modify the behavioral specification until it matches the desired performance. Once a behavioral specification has been established, some form of synthesis is employed to generate a description of the structural specification. Synthesis in this context is performed by first deriving a netlist from the behavioral specification then optimizing the derived netlist into its implementation form. The synthesis may be performed manually or automatically. The structural specification is a description of the actual hardware component to be realized. At this point, the behavioral and structural specifications are simulated to generate test vectors for comparison. Notice that the simulation of the behavioral specification performed at this point is in addition to the one performed earlier for examining the behavioral specification. This process of ensuring conformance is referred to as validation.

From the implementation specification, layout of the integrated circuit is performed, again, through synthesis. As before, the synthesis may be either manual or automatic. From the generated layout-specification, a transistor netlist may be generated for use in a switch-level simulator (Terma 80). The results produced by the switch-level simulator are then compared against the results of the structural-specification simulation. Once the layout-specification is shown to conform to the structural specification, it is transformed into a format that may be sent to a fabrication service for production of the integrated circuit[3].

Once the component has been fabricated it must be tested for fabrication flaws. A set of test vectors is run on the integrated circuit. The test results are compared against the results of

[2]See Appendix A for a discussion of behavioral and structural specifications.
[3]For the purpose of this presentation, the CALTECH Intermediate Format (CIF) was chosen.

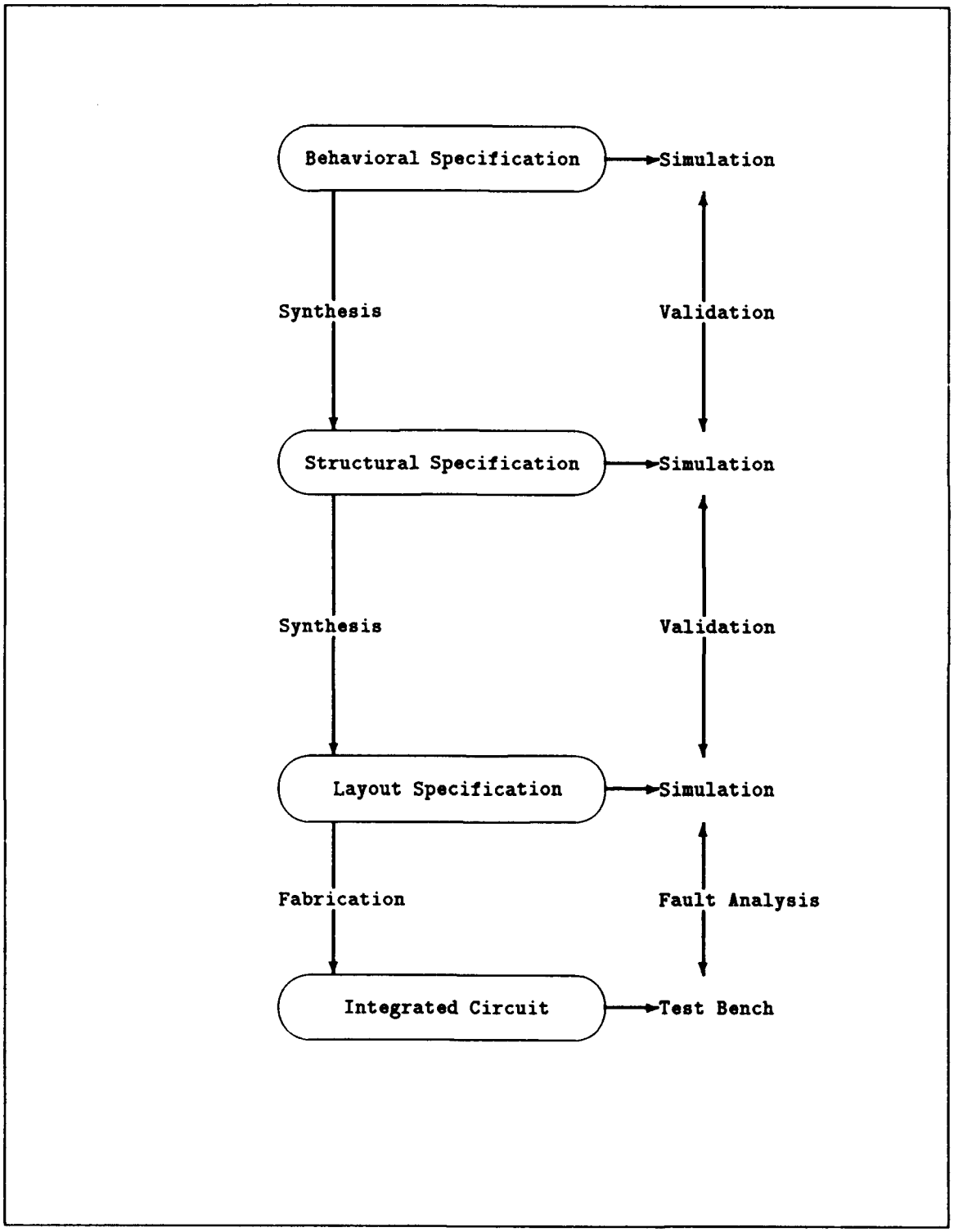


Figure 1. General CAD Development of Hardware.

simulating the structural specification. Should conformance exist between the integrated circuit and the structural specification, the integrated circuit is assumed to be correct.

Figure 2 is a diagram of a general CAD environment that includes the use of formal hardware-verification. With the use of formal hardware-verification, it is no longer necessary to simulate the behavioral specification for the purpose of generating test vectors for the structural specification. The simulation of the structural specification is necessary only for comparison against the final fabricated component.

Problem

Using simulation to validate compliance between a structural specification and its layout specification is no longer acceptable. Designs built today have increased in complexity well beyond the designs built through breadboarding. Though simulation was sufficient for small 1,000-transistor designs, designs are currently being constructed on the scale of 100,000 to 1,000,000 transistors. To help understand the complexity of the problem, consider a 32-bit adder.

A hardware structure typically found on an ASIC today is a 32-bit adder. Such an adder may be implemented in a number of different ways (Weste 85:310-331). Regardless of the implementation, for input there are 32 bits for one operand, 32 bits for a second operand, and a carry bit. The total number of input-bits for a 32-bit adder is thus 65. For an exhaustive simulation[4] at least 36,893,488,147,419,103,232 test vectors would be required. If we assume that a simulator running today could handle 1,000 test vectors per second, the system would complete the simulation within 1,169,884,834 years; however, simulation to this extent would still not guarantee equivalence between the structural specification and the layout specification. All that is guaranteed is that both the structural specification and the layout specification are equivalent for the given test-vector sequence. Therefore, performing exhaustive simulation is ineffective, even for simple designs.

[4]The next chapter provides a case where exhaustive simulation for an assumed combinational circuit may not be sufficient, requiring even more test vectors than shown here.

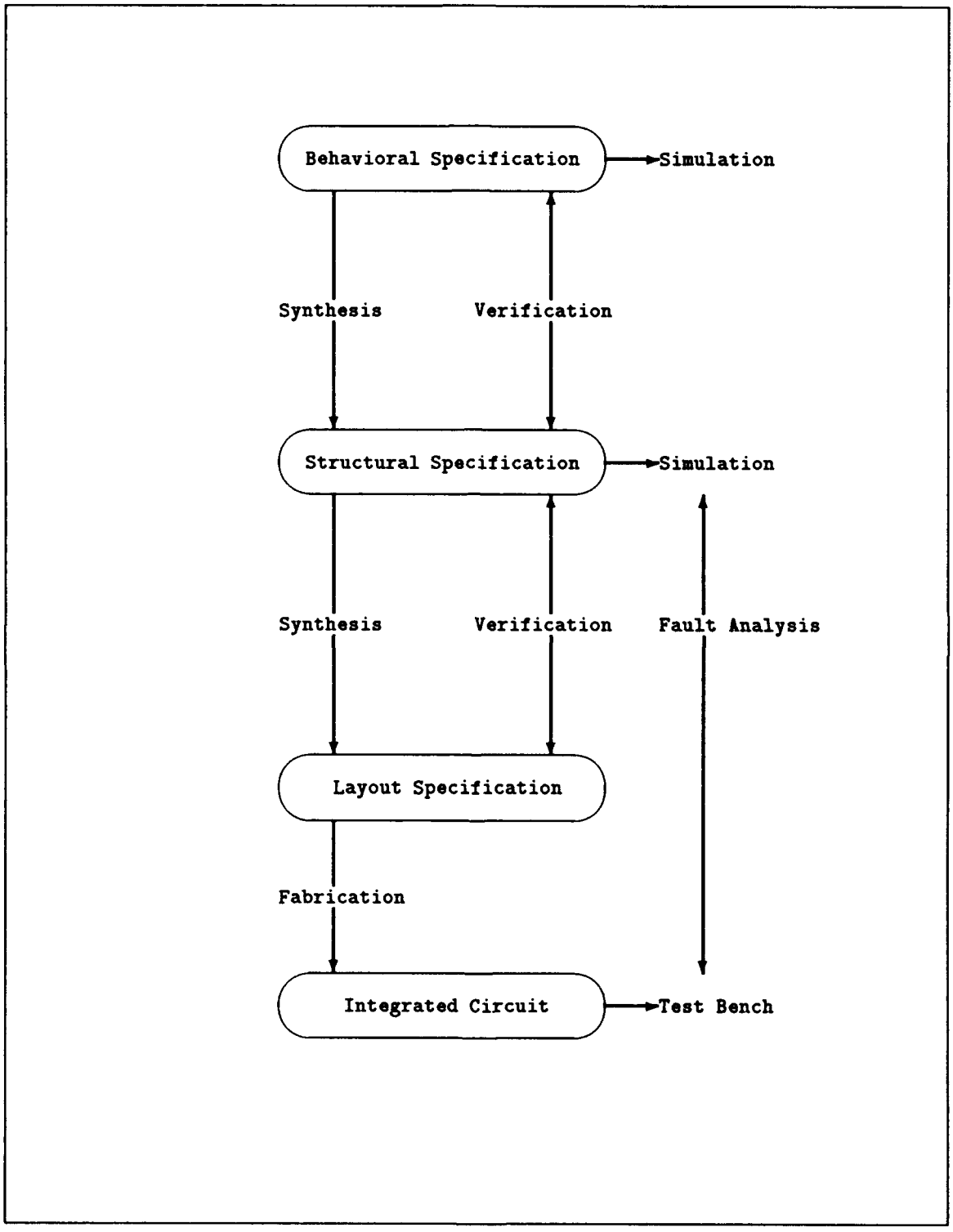


Figure 2. General CAD Development Using Formal Hardware-Verification.

Solution

One way to overcome the deficiencies of simulation is through logic extraction. Logic extraction ensures that a layout-specification is equivalent to a structural specification. Further, the time that it takes to verify the equivalence between a structural specification and a layout-specification of designs like the 32-bit adder is several seconds of total CPU time. As a result of the research reported here, a formal basis for logic extraction is presented, logic extraction of large VHSIC-class chips is possible, and the modest CPU/memory requirements of logic extraction make verification of 1,000,000-transistor designs a reality. Thus, the objective of this research is to establish a formal definition of logic extraction, discuss properties of logic extraction as they relate to formal hardware-verification, and demonstrate that logic extraction is practical for VHSIC-class designs.

Overview

GES is a collection of programs written in Prolog. Prolog is used as the vehicle for investigating and implementing logic extraction, since logic programming is directly implemented. A logic program is a collection of rules, where a rule has the form,

$$A \longleftarrow B_1, \ldots, B_n$$

and $n \geq 0$ (Sterl 86:8-15). GES consists of several Prolog programs that perform the following functions.

1. *ges* - the logic extraction system
2. *sim2pro* - a filter for translating a .sim transistor netlist to *ges* format
3. *vhdl2ges* - generates *ges* from a hierarchical VHDL structural description
4. *flatten* - flattens a hierarchical VHDL structural description to a netlist of its lowest-level components

5. *vhdl2ecpv* - generates *ges* from VHDL for extracting critical paths in a netlist generated by *flatten*

6. *vhdl2ecpl* - generates *ges* from VHDL for extracting critical paths from layout

7. *ges2vhdl* - generates VHDL structural description from an extracted component netlist

8. *geng2v* - generates a *ges2vhdl* tool from a collection of hierarchical VHDL structural descriptions

This dissertation will focus on proving several properties about *ges*[5]. These properties are correctness, completeness, and termination. By demonstrating correctness, we show that any circuit successfully extracted by *ges* is indeed the circuit that was intended to be built. By demonstrating completeness, we would like to show that if a circuit was built according to its structural specification, *ges* would succeed in extracting it; however, we will show that this is not always the case for logic extraction. We will also show that logic extraction, through *ges*, is guaranteed to terminate. Finally, we will show that pin-to-pin critical path analysis is possible within the context of logic extraction.

[5]GES encompasses all of the Prolog programs enumerated here. *ges* is the Prolog program that performs the logic extraction.

II. Validation, Synthesis, and Verification

Several approaches to ensuring the conformance of hardware implementations to hardware specifications are reviewed in this chapter. These approaches are generally referred to as validation, verification, and synthesis. We will show that validation is inadequate for today's designs. We will also show that the assertion of "correct by construction" made by designers of synthesis tools is not true. All of these approaches are related to this work and help to place this work in perspective. Before examining these different approaches, the generic design process presented in Figure 1 and Figure 2 should be reviewed.

Validation Techniques

Validation is concerned with demonstrating the functionality of a given circuit for a selected set of input stimuli and output responses. Stated another way, validation is used to demonstrate, through a collection of results or test vectors, the compliance of one hardware description with another hardware description. Simulation is also used as a name for the process of validation. Exhaustive simulation is not feasible for any but the simplest of digital designs. If we consider only a 32-bit register, there are over four billion possible output responses for any one given input (Barro 84:438). The process of simulation is NP-complete and in some cases may not be exhaustive.

Problems with design validation are not limited to its intractability alone. Two basic types of simulation, event-driven and switch-level, are also prone to complications due to the nature of the simulation cycle. Switch-level simulators, for example, are generally used to perform simulation of the mask layout description as a means of validation (Terma 86). This type of simulation is based on a state model. As such, the simulation cycle is based on propagating logic values through a circuit network until a steady state is reached. For combinational logic circuits, this type of simulation model does not present any difficulty. For sequential circuitry and systems with oscillating feedback loops, simulator problems are generated through nonconverging circuit

configurations. Certain other sequential circuits can also introduce race conditions that cannot be handled. Circuits that have oscillators as part of their normal makeup never converge to a steady state value once they are set to oscillate. Since the circuit never converges to a steady state, simulation using switch-level simulators is not practical.

Using switch-level simulators to identify errors in a circuit may also prove difficult. Consider the circuits shown in Figure 3(Bryan 87). Should the sequence of test vectors for (A,B) be chosen as ((0,0), (0,1), (1,1), (1,0)), the result for (A,B,Out) would yield the sequence ((0,0,1), (0,1,0), (1,1,0), (1,0,0)). The same sequence would be seen for both the correct NOR-gate implementation and the flawed NOR-gate implementation. The capacitive storage on the flawed NOR-gate circuit allows for a logical 0 on Out when (A,B) is set to (1,0). However, had the sequence for (A,B) been chosen as ((0,0), (1,0), (1,1), (0,1)) the flaw would have been detected for (A,B,Out) as (1,0,1). This example demonstrates that flaws in a combinational circuit may not be found even when an "exhaustive" sequence is used to simulate the circuit's behavior.

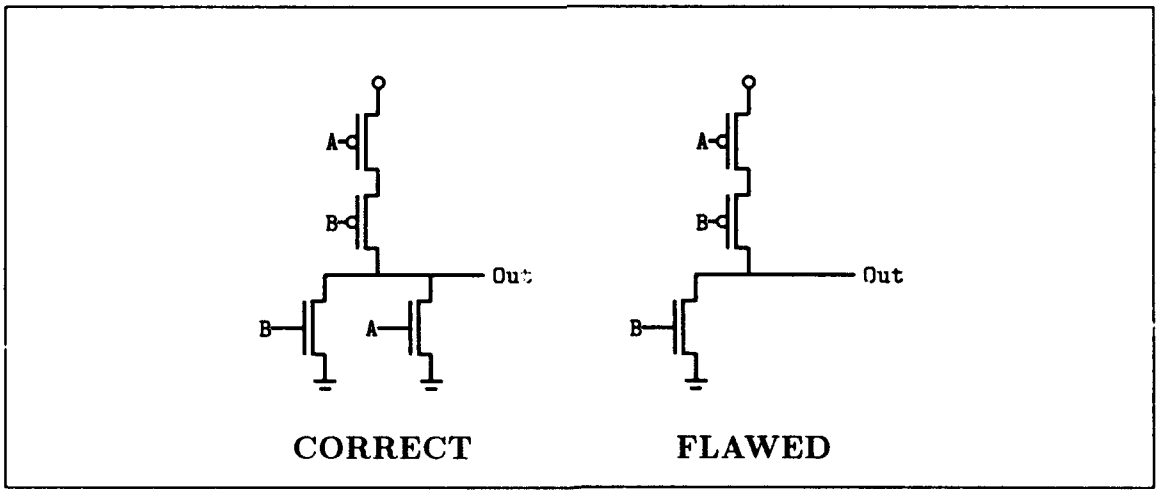


Figure 3. Circuit Diagrams for a NOR-Gate and Flawed NOR-Gate.

Event-driven simulators are based on propagating signals using time-value pairs (IEEE 87). This type of simulation allows for delay to be considered. Sequential circuits that oscillate, then, do

not cause undeterminable results. This type of simulation usually is performed on digital circuits at the gate level.

Synthesis

The purpose of this section is to discuss problems of using automatic synthesis alone in the hardware design process. Ideally, synthesis translates a register-transfer level (RTL) description (structural specification) directly to a layout-specification. However, practical synthesis is a two-step process involving translation from a RTL description to a component netlist followed by an optimization step. The translation portion of synthesis maps a RTL description of a design in a hardware description language[1] to a gate-level netlist (deGeu 89:27). Afterwards, optimization of the gate-level netlist is performed. The optimization step may be based on an optimization process using the Quine-McCluskey (McClu 56) method or using a rule-based substitution process.

The first problem with synthesis concerns the completeness of the required design specification. The behavior of the design must be described completely in order for synthesis tools to generate a hardware description. If we assume a specification that describes the output condition when A=B=1 to be 1 and the output condition for A=B=0 to be 0 without further information, synthesis cannot be performed. We may assume don't care conditions for the other two situations; however, this condition must be explicitly stated. The designer may be required to fully describe a given design even when doing so may be highly inconvenient.

Synthesis is a highly complex process. To further complicate the problem, VHDL is composed of many procedural and declarative language constructs. The complexity of synthesis and the many features of VHDL can contribute to generating inadvertant errors during the mapping to a gate-level netlist, optimization, or the mapping to a layout specification (Devad 88:182). Therefore, synthesis needs to be checked by a verification system to ensure conformance.

[1] VHDL is the standard hardware description language used for a RTL description.

Another problem with synthesis is the limited set of design solutions that are provided. Generally, a fixed generation-pattern from a RTL representation in VHDL to a gate-level netlist exists for a given language construction. For example, a case statement in VHDL may be mapped directly to a series of multiplexers in hardware. Some synthesis tools provide the means to make space versus area tradeoffs during optimization (deGeu 89:29). However, these solutions are equivalent solutions. Other solutions may exist that meet the criteria of the behavioral specification, but are not equivalent. For the purpose of illustration, assume that a designer wishes to incorporate a full adder into a design. Assume also that previously fabricated components exist, but with two full adders on a chip. The chip with two full adders would suffice for the needs of the designer; however, the synthesis system would tell the designer to design a new component comprising one full adder. This problem limits the designer to a confined solution space when investigating alternative solutions might yield better designs.

The level where synthesis is performed is important. When an expert manually synthesizes a structural specification from a behavioral specification, some mental rechecking of the behavioral specification is performed. Flaws or poor assumptions made in the behavioral specification are sometimes found while the expert is exploring the solution space of the structural specification. Synthesis, however, doesn't provide this opportunity to reflect on the original behavioral specification. Thus, the flaws and poor assumptions in the behavioral specification are incorporated into the final product.

Synthesis systems are generally based upon Boolean manipulation techniques. If the design specification is not based on {0,1} then the Boolean manipulation techniques used in the synthesis approach will not yield an implementation description that may be compared to the behavioral specification. Assume that the behavioral specification for a given design is

$$out = (xy \vee xz \vee y'z). \tag{1}$$

A synthesis system would perform optimization on Eq 1 yielding the following transformation.

$$(xy \vee xz \vee y'z) \mapsto (xy \vee y'z) \tag{2}$$

Through synthesis, the expression on the left of Eq 2 may be seen to be equivalent to the expression on the right. The unannounced assumption made by the synthesis system in this case is that the above expression is true for a Boolean algebra. However, should we choose the case where the set of possible values used is $\{0,X,1\}$[2] great difficulty arises in generating comparable simulation results before and after synthesis.

Shown in Figure 4 is an uncomplemented distributive lattice[3] [4] and a collection of operator tables defined for {0,X,1}. The uncomplemented distributive lattice and the collection of operator tables are not the same and differ through the interpretation of complement. For a true complement to exist, the following system of equations must be satisfied (Rudea 74:3-4) (Donne 68:101).

$$\begin{aligned} u \wedge u' &= 0 \\ u \vee u' &= 1 \end{aligned}$$

A true complement does not exist for the operator tables since the system of equations cannot be satisfied for $u = X$. Attempting to force a complement operation for {0,X,1} only succeeds in generating problems in simulation. An example can be constructed to illustrate this problem.

For the previous transformation, Eq 1, assume $x = z = 1$ and $y = X$. The result for both equations of the transformation would then be the following.

$$(xy \vee xz \vee y'z) = 1 \tag{3}$$

[2]This set is from the current EIA modeling standards for VHDL (EIA 89). Such a set of values is used to perform hazard analysis in CMOS circuits

[3]A value system described in this manner is not unusual and was first proposed by Lukasiewicz in 1920 (Resch 69:22).

[4]A discussion on lattices may be found in (Donne 68). In particular, a Boolean algebra is a complemented distributive lattice (Donne 68:55-59,224-249). The relation between a Boolean algebra and the postulates that define a complemented distributive lattice are stated explicitly in (Rudea 74:4).

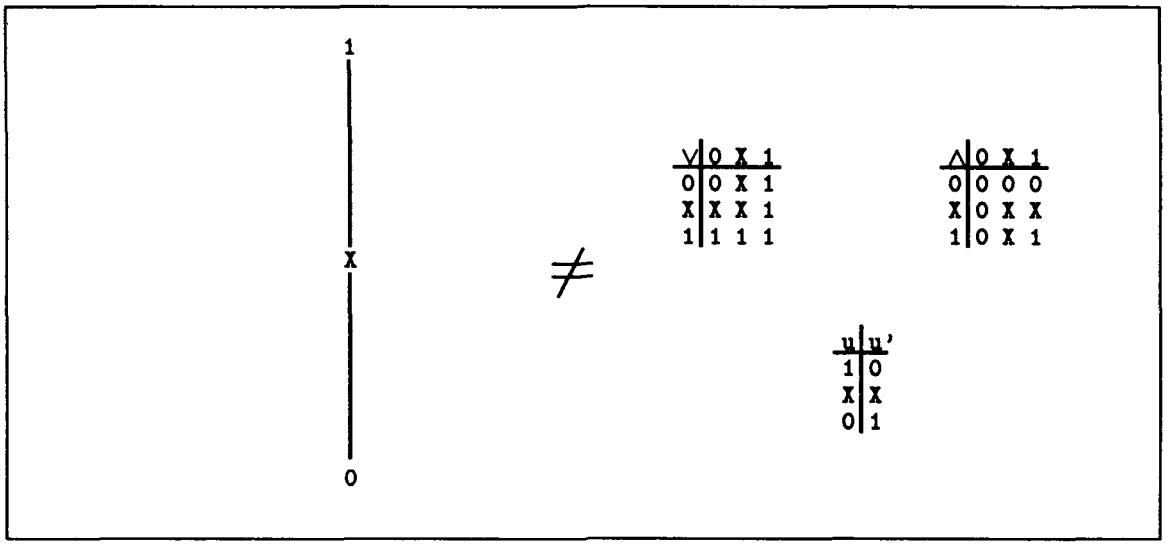


Figure 4. Lattice and Operator Tables for {0,X,1}.

$$(xy \vee y'z) \;=\; X \tag{4}$$

Eqs 3 and 4 have different results leading the user to believe that the synthesis tool has failed. However, the result of synthesis using consensus to absorb the xz term is valid for a Boolean Algebra. This illustration demonstrates the problem of comparing results through simulating a synthesized structural specification from a behavioral specification when a many-valued logic system is used.

Simulation using a many-valued logic system is not the only problem encountered when synthesis is used in Eq 2. The designer may desire to have the xz term included. Without the xz term, a spike might be induced into a circuit when transitioning from the xy term to the $y'z$ term. Using synthesis in some situations may produce large combinational logic circuits with dangerous transient responses to certain input stimuli.

Although there are problems with synthesis, its use is beneficial in some cases. In situations where certain designs described by a hardware description language are easily synthesized, synthesis can provide better hardware solutions than can humans. When working with large circuit designs, humans can lose attention while optimizing a circuit. Synthesis tools, however, work as vigorously

to optimize the last portion of a circuit as they do the first part. Most synthesis tools also perform some self-verification of the hardware they have generated, to help reduce the possibility of introducing errors. Synthesis requires reasonable human guidance and a verification system for proper operation.

Verification Methods

As opposed to validation, verification is the process of proving compliance between one hardware specification and another hardware specification. Verification methods are based on a sound mathematical foundation. Methods for formal hardware-verification range from automatic (Boyer 79) to manual (Gordo 89). The Boyer-Moore method performs most of its theorem proving through induction, whereas HOL is open to theorem proving through many techniques. One of the more common techniques of theorem proving in HOL is through rewriting of goals and dividing goals into a conjunction of subgoals.

The capacity to perform formal hardware-verification is based upon the ability to express hardware descriptions in a theorem form. Some formal hardware-verification systems require that a description of the hardware be in a specification language. The specification language may either be the language of the proof system or some hardware description language that can be translated into the language of the proof system.

The relationships among the behavioral specification, structural specification, and layout specification, are usually characterized as follows:

$$\textit{Structural Specification} \Rightarrow \textit{Behavioral Specification} \tag{5}$$

$$\textit{Structural Specification} \Leftrightarrow \textit{Layout Specification} \tag{6}$$

Relation 6 is necessary to ensure that the actual hardware and documentation match explicitly. Contrary to intuition[5], Relation 5 expresses the logical requirement that the structural specification is within the domain described by the behavioral specification.

The implicative relation between a behavioral specification and a structural specification can be shown through several examples. For the first example, assume the domain of discourse to be the set of integers. Considering a behavioral specification, $x^2 = 25$, and two structural specifications, $x = 5$ and $x = -5$, we have

$$(x = 5) \Rightarrow (x^2 = 25)$$

$$(x = -5) \Rightarrow (x^2 = 25).$$

The equation (structural specification) $x = 5$ is a solution (implementation) of the equation (behavioral specification) $x^2 = 25$. In Appendix C is a digital-hardware example of the implicative relation between several structural specifications and a behavioral specification.

Further information regarding formal hardware-verification techniques may be found in several sources. An extensive survey of formal hardware-verification is in (Camur 88). Some of the most commonly referenced methods include Higher-Order Logic (Gordo 89), Boyer-Moore (Boyer 79), and TEMPURA (Moszk 86). A discussion on temporal logic approaches is presented in (Galto 87). A tutorial for HOL has recently been published (Gordo 89). Furthermore, a theoretical discussion of HOL may be found in (Gordo 88). A demonstration of how HOL is used in formal hardware-verification may be found in Appendix C.

[5]Those initially exposed to simulation as a form of design validation tend to see this relation as *Behavioral Specification* $\Rightarrow$ *Structural Specification*. This bias is brought about by the thought that whatever stimulus-response patterns result from the behavioral specification must also result from the structural specification.

Summary

The process of simulation is neither correct nor complete. A circuit may not match a design, but simulation may lead the designer to believe the design is constructed in compliance with its structural specification. Additionally, a circuit may comply with its structural specification, but problems with the simulator paradigm may prevent demonstration of proper performance. Despite these shortfalls, simulation is still necessary for generating test vectors for testing fabricated components.

Synthesis is not always sufficient for guaranteeing correct designs and may produce unexpected results. Thus, synthesis requires the use of formal hardware-verification to ensure the results of synthesis are correct. In contrast to design validation, formal hardware-verification can verify a design for all possible inputs and outputs in a tractable manner (Barro 84:438).

We have shown that a structural specification is a solution to a behavioral specification, but not necessarily a unique one. Since a structural specification is a reflection of the actual hardware, it must match the layout specification explicitly. Formal hardware-verification methods should be used to show that the layout specification is equivalent to the structural specification. As a formal method, logic extraction shows equivalence between the layout specification and the structural specification.

III. Logic Extraction, VHDL, and Transistor Netlists

A Survey of Logic Extraction

Logic extraction has been attempted by several researchers prior to this research. Previously, logic extraction was viewed as simply a matter of matching a few subcomponents to a template and replacing them with a single component; however, problems were encountered with component connectivity when using this simple approach. Additionally, the size of the circuit that could be extracted was less than 10,000 transistors in size.

An informal description of logic extraction through GES is first presented. Afterwards, descriptions of previous approaches are provided. As each approach is discussed, the inherent problems with each approach are identified.

Logic Extraction through GES Three functions are performed in the extraction process shown in Figure 5. These three functions are identifying the subcomponents that form the component, checking that internal nodes do not have external connections, and checking for internal-external connection inconsistencies. By extraction, the appropriate components and interconnections are identified. When the components are identified, additional checks are necessary to ensure that different variable labels contain different values[1]. Checking that the internal connections of a component do not connect with another component external to the component being extracted is important. From Figure 5, the node named **INTERN** "disappears" from the overall circuit once the D-latch is extracted. If the node named **INTERN** is connected to some other component external to the D-latch, connectivity information is lost.

Prolog vs. Forward-Chaining Expert Systems Rule-based methods implemented in forward-chaining systems like OPS5 (Spick 85), OPS83, or CLIPS (Yaros 89) suffer from two problems. The first problem is inherent to the extraction process. The second problem is inherent to

[1] Within Prolog, different variable labels are allowed to contain the same value. However, when using different variable labels for hardware design, different electrical connections are implied.

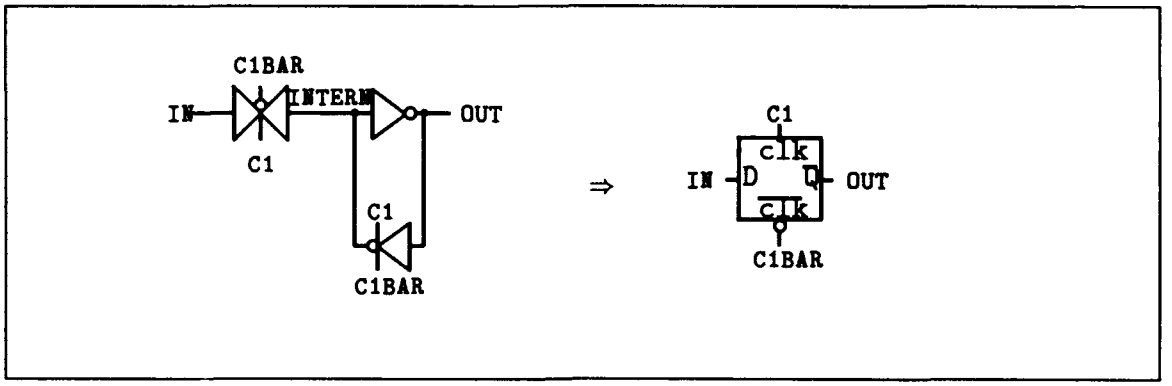


Figure 5. Logic Extraction.

forward-chaining systems. The types of forward-chaining systems that fall within the realm of this discussion are shown in the pictorial representation of Figure 6. The forward-chaining system is divided into parts. The first part consists of the rules. Each rule has a left-hand side, **LHS**, and a right-hand side, **RHS**. The **LHS** is a set of conditions that must be met in order to carry out the actions in the **RHS**. The working memory contains the facts and context of the forward-chaining system. The context is a stack of all partially matched and fully matched rules. An iterative process of matching, conflict resolution, and acting is carried out until there are no more fully matched rules to act on. The conflict resolution portion of the iterative process determines which fully-matched rule to act on.

Figure 7 is an example of what appears to be a **NAND** gate; however, an internal connection, **Int**, to another component, **COMP**, suggests otherwise. Before extracting a component, its internal connections must be checked to ensure they are not connected to the external connections of the component being extracted ("local" connectivity) and to ensure they are not connected to another component ("global" connectivity). Once the component is recognized and extracted, the internal node disappears, thus connectivity information is lost. Local connectivity is easy to check within an extraction rule, since all of the connection information is available. Global connectivity is more difficult to check. It is not readily apparent how rules in forward-chaining systems may

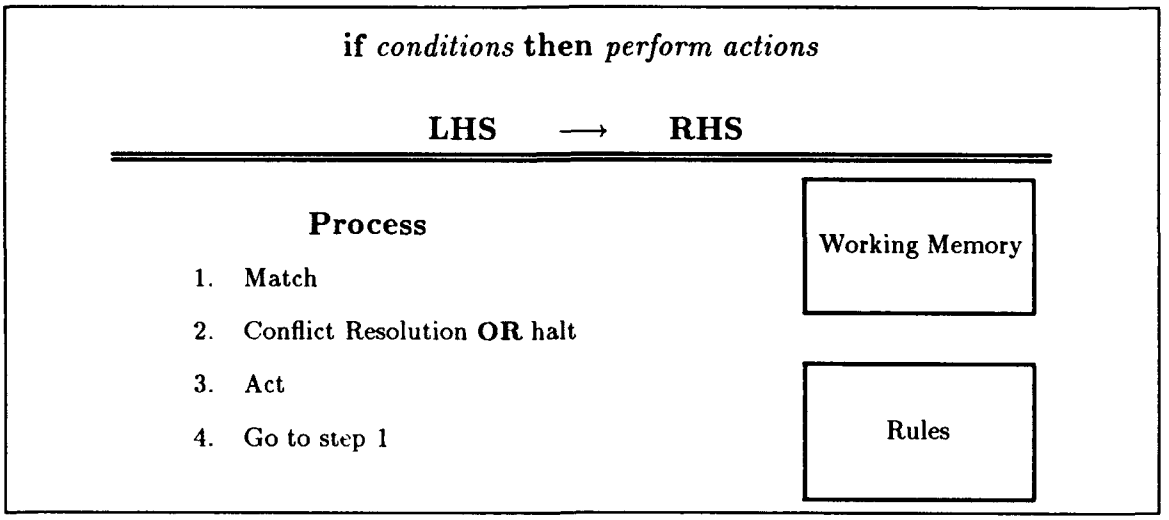


Figure 6. Forward-Chaining System.

be constructed to solve the internal-global connectivity problem. Such rules for internal-global connectivity checks are not addressed in (Spick 85) (Yaros 89).

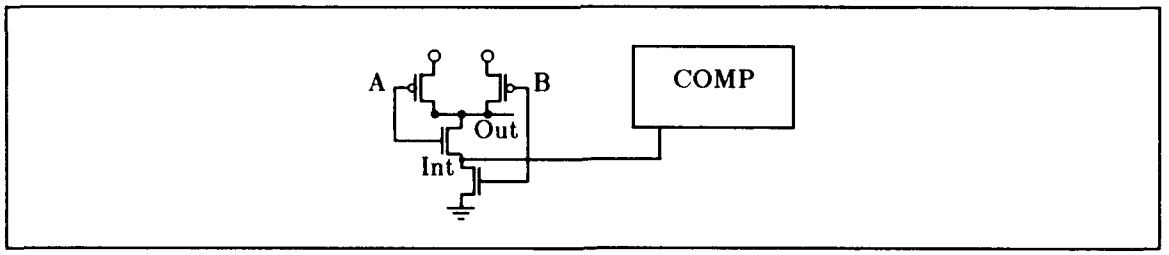


Figure 7. Possible NAND Gate.

Secondly, forward-chaining systems are best-suited for heuristic solutions to problems of an intractable or diffuse nature. The "working memory" of an expert system is one element where impact on performance is observed. Working memory contains both the facts (in this case, a transistor netlist) and the context of the system (a stack of fully matched and partially matched rules). A large number of additions or deletions of facts can result in costly memory management overhead. Stored within the context of working memory is a list of matched and partially matched rules. Having several rules governing component configurations from transistors among a fact-base

of several thousand transistors can result in the context area of working memory exceeding the capacity of the system. A CLIPS implementation of logic extraction (Yaros 89) developed along the lines of an OPS5 implementation of logic extraction (Spick 85) demonstrated that a system of several matching-rules (gate-types) and 2,000 transistor-facts exceeded 100 MegaBytes of available memory during execution.

Aside from the large memory problems, it turns out that conflict resolution is of little importance. The component netlist may be extracted simply by applying the rules sequentially. The overhead involved in resolving conflicts between rules ready to fire is an unnecessary expense. Further, the inability to directly control the rule-firing order makes the logic extraction process difficult to fine-tune for efficiency in an expert system environment.

Prolog vs. Other Languages Prolog was chosen as the implementation language over procedural languages and Lisp for several reasons. The logic extraction process involves searching and pattern matching. Prolog is naturally suited to searching and pattern matching. Expressing the logic extraction process in Prolog allowed for rapid-prototyping of ideas. Seventy lines of Prolog code, easily developed, implemented a portion of the extraction process being performed by a C-code implementation of over 5,000 lines (Linde 88). Further, reliable Prolog implementations exist today (Quint 88). Finally, there is an accepted standard for Prolog (Clock 87b).

Summary Work with logic extraction has been reported in (Spick 85) (Boehn 88). Other work that performs comparisons of transistor networks to their original structural descriptions (either an HDL or schematic) do so by graph-based methods (Ebeli 83) (Takas 88), rule-based methods (Takas 88) (Papas 88), or other methods (Boehn 88) (Takas 88) (Spick 83). Though one method does extract some timing information at the gate level (Boehn 88), none performs any type of critical path analysis in conjunction with the extraction process. A report on critical path analysis within the extraction process is in (Dukes 91a).

The Representation of Hardware Structure Through VHDL

VHDL Syntax This section details the VHDL language constructs accepted by *vhdl2ges*. Several examples of acceptable structural VHDL models are provided. The VHDL language constructs are taken from the Syntax Summary of (Dukes 91b:5). At a minimum, the VHDL description must contain the following.

entity identifier **is**
  *formal*_port_clause
  end *entity*_simple_name ;

architecture identifier **of** *entity*_name **is**
  component identifier
    *local*_port_clause
    end component;
  begin
  *instantiation*_label :
    *component*_name port_map_aspect ;
  end *architecture*_simple_name ;

Below is an example of a VHDL description conforming to the above description.

```
entity comp is
  port (A : in bit);
  end comp;

architecture structure of comp is
  component sub_comp
    port (A : in bit);
    end component;
  begin
  sub_comp00 : sub_comp port map (A);
  end structure;
```

Additional VHDL language constructs supported are shown below.

signal identifier_list : subtype_indication;

alias identifier : subtype_indication **is** name;

All other VHDL language constructs are ignored.

Properties of Structural VHDL There are several properties and relations expressed by a structural VHDL description. Those properties and relations concern the component-to-entity relation, properties of signal names, and properties of aliases. Further, there are the restrictions that at least one signal must be in the port and there must be at least one instantiated component in the architecture body.

The relation between the entity and its respective components is a one-to-many relation. This relation is shown in Figure 8. Importantly, this relation is bidirectional in that an entity may be decomposed into a collection of components or a collection of components may make up an entity. In the case of logic extraction, the emphasis is on finding a collection of components that make up an entity.

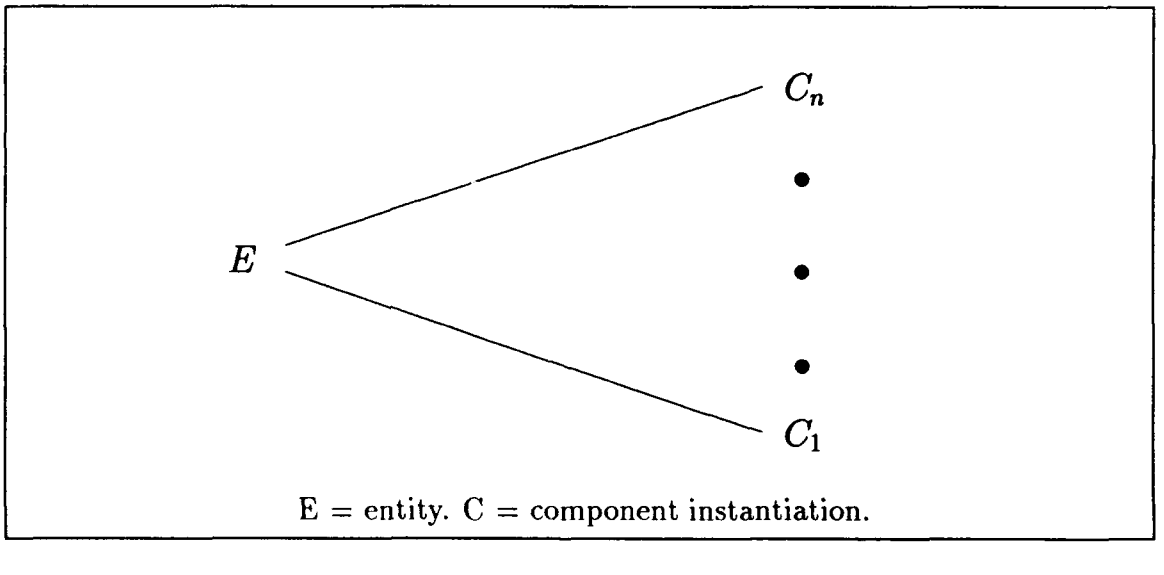


Figure 8. Relation Between an Entity and Components.

The port of an entity lists the signals through which the entity communicates to exterior components. The port forms a boundary, isolating the interior components. The signals declared in the declarative area of the architecture imply "wires" through which the components of the interior of the entity communicate. Some important distinctions between signals of the port and

signals of the architecture's declarative area arise. For clarity of the discussion, we will use $signals_e$ to denote signals of the port and $signals_a$ to denote signals of the architecture declarative area.

For the signals of $signals_a$ and $signals_e$, we will informally describe a logical property called $not\_connected(signals_a, signals_e)$. We may also refer to the logical property $not\_connected(signals_a, signals_e)$ as **not_connected/2**[2]. The properties of **not_connected/2** are as follows.

1. None of the signals of $signals_a$ may be a member of $signals_e$.
2. Every signal of $signals_a$ is unique.

The two properties of **not_connected/2** reflect the semantics of VHDL. The first property of **not_connected/2** enforces the requirement in VHDL that a signal may only be declared in the port of the entity or in the architectural declarative part. The second property of **not_connected/2** represents that fact that each signal declared in the architectural declarative part represents a unique wire. However, the signals of the port map may be interconnected when the component is instantiated at a higher level in a VHDL structural specification.

As for the visibility of the signals of $signals_a$, a property $find\_anomaly(component, signals_a)$ will be defined. The property, $find\_anomaly(component, signals_a)$, explicitly represents the fact that none of the signals of $signals_a$ may be connected to components outside the entity under consideration. In essence, $find\_anomaly(component, signals_a)$ expresses the confinement of the scope of the signals of $signals_a$ to the interior of the architecture of the entity under consideration. Implicitly, the signals of $signals_e$ may be seen on the outside and inside of the entity.

Aliases in VHDL provide a basis for renaming signals of $signals_a$. Any time an alias is encountered, it is replaced with the appropriate signal of $signals_a$ or $signals_e$.

[2] A Prolog program is identified by its functor and arity as *functor/arity*. The functor is the name of the Prolog program and the arity is the number of parameters passed to the Prolog program.

Transistor Netlist Representation

GES does not extract components directly from a layout specification. GES performs logic extraction on a transistor netlist derived from a layout specification. The transistor netlist may be generated from a layout specification from one of several CAD tools that already exist for this purpose. This section describes the input format for a transistor netlist. Also presented is a mapping from a transistor-netlist format produced by a CAD tool to the transistor-netlist format used for logic extraction.

Generating Transistor Netlists from Magic One form for a transistor netlist is that described in (Terma 86). This form was chosen since it was derived from the mask layout form of *magic* used in (Calif 86). One process of generating the transistor netlist begins in *magic* by using the **:cif** command. The **:cif** command in *magic* produces a mask layout file in CIF[3]. Afterwards, *mextra* reads in the CIF file and produces a transistor netlist file. The record format for a transistor is

```
type gate source drain length width xpos ypos
```

where **type** is one of e, p, or n for enhancement mode transistor, p-type transistor, or n-type transistor, respectively. The second through fourth fields, **gate**, **source**, and **drain**, describe three of the four terminals of the MOS transistor used. The bulk (or substrate) is assumed to be biased correctly and is not included. The fifth and sixth fields, **length** and **width**, describe the channel length and channel width of the MOS transistor. The seventh and eighth fields, **xpos** and **ypos**, are the location coordinates of the transistor.

An alternate route for extracting a transistor netlist from a *magic* layout also exists. The **:extract** command in *magic* creates a hierarchical form of the layout, in **.ext** format, currently

[3]CALTECH Intermediate Format (CIF) is one layout format used. Another common layout format is GDS II.

residing in *magic*. A series of extraction files is created that reflects the cell hierarchy used in *magic*. A tool called *ext2sim* is then used to generate the transistor netlist form.

Converting Transistor Netlists to Prolog Clause Form If the *mextra* tool is used on a CIF file, the transistor netlist will be created with the n-type and p-type transistors described as e and p for their types, respectively. If the *ext2sim* tool is used, the transistor netlist will be created with n-type and p-type transistors described as n and p for their types, respectively. Thus, a transistor generated from *mextra* as

```
e a_XNOR_c#17 1520 GND 300 1200 706200 -20550
```

would appear in Prolog clause form as

$$n(nA\_XNOR\_C17, n1520, ngnd, 300, 1200, 706200, -20550).$$

A transistor generated from *ext2sim* as

```
p 20/4_1/A_in_nand Vdd 20/4_1/probe 300 1200 12172 101
```

would appear in Prolog clause form as

$$p(n204\_1A\_IN\_NAND, nvdd, n204\_PROBE, 300, 1200, 12172, 101).$$

The general transistor netlist format in Prolog is shown below.

$$type_P(gate_P, source_P, drain_P, xpos, ypos).$$

The following mappings are used:

$$type \mapsto type_P$$

$$
\begin{aligned}
p &\mapsto p \\
n &\mapsto n \\
e &\mapsto n
\end{aligned}
$$

$$
\begin{aligned}
gate &\mapsto gate_P \\
node &\mapsto nNODE \\
Vdd &\mapsto nvdd \\
GND &\mapsto ngnd \\
Gnd &\mapsto ngnd
\end{aligned}
$$

$$
\begin{aligned}
source &\mapsto source_P \\
node &\mapsto nNODE \\
Vdd &\mapsto nvdd \\
GND &\mapsto ngnd \\
Gnd &\mapsto ngnd
\end{aligned}
$$

$$
\begin{aligned}
drain &\mapsto drain_P \\
node &\mapsto nNODE \\
Vdd &\mapsto nvdd \\
GND &\mapsto ngnd \\
Gnd &\mapsto ngnd
\end{aligned}
$$

where

```
node ::= letter_number_specialcharacter{letter_number_specialcharacter}

letter_number_specialcharacter ::= letter | number | specialcharacter

letter ::= upper_case_letter | lower_case_letter
```

and

```
nNODE ::= n{[underline]upper_case_letter_number}

upper_case_letter_number ::= upper_case_letter | number
```

The set *specialcharacter* $= \{@, \#, \%, *, (,), /, [,], ', ", `\}$ is not necessarily a complete one since *magic* allows labels to consist of a large number of different special symbols. The process *node* $\mapsto$

nNODE drops *specialcharacter* and translates `lower_case_letter` to `upper_case_letter`. The mapping to upper case letters is necessary to overcome the case sensitivity of *magic* and Prolog when generating case-insensitive VHDL code.

IV. A Formal Approach to Logic Extraction

The extraction process is a method of proving and determining the existence of higher level constructs from existing lower-level ones. By seeing a relation between component definitions and extraction rules, we may demonstrate that the extraction process is a form of hardware-verification. In fact, the highest level Prolog rule may also be the final goal to be achieved in an extraction process whereby only one component is left over after the entire extraction process has run its course.

Past attempts at logic extraction have failed in their expression of the essence of logic extraction. Ensuring that certain properties (i.e., protecting local and global connectivity[1]) exist has not not been addressed. Had a formal approach to logic extraction been previously attempted, these properties may have been discovered.

A formal definition of logic extraction will be presented in terms of logic-programming. Afterwards, the important properties of a component's description as they relate to VHDL will be proved. Finally, properties concerning correctness, completeness, and guaranteed termination will be presented.

Defining Lists

As a matter of convenience, a logic-programming representation in Prolog is used to describe the logic extraction system. Using Prolog is proper for describing formal methods and executing them (Wing 90b:15). Therefore, developing a unique syntax and semantics for the logic extraction system is unnecessary. The structures that are used are assumed to be finite. The various components to be used for logic extraction will be described first.

A property called $not\_connected(signals_a, signals_e)$ was stated previously in Chapter 3. Reiterating the definition for this property we have,

[1] The discussion on local and global connectivity is in Chapter 3.

not_connected($signals_a, signals_e$) $\Leftarrow$ *None of the signals of* $signals_a$ *may be a member of* $signals_e$. *Every signal of* $signals_a$ *is unique.*

Prior to presenting a Prolog definition for **not_connected/2** the representation for $signals_a$ and $signals_e$ should be described.

Previously, $signals_e$ was described as the signals of the port and $signals_a$ was described as the signals of the architecture. To make $signals_a$ and $signals_e$ useful to Prolog, we will choose a list representation for both. Furthermore, each signal of $signals_a$ and $signals_e$ will be a Prolog atom[2]. In Prolog, **not_connected/2** may be expressed as

```
not_connected([Signal|ResetOfSignalsA],SignalsE) :-
  not_member(Signal,ResetOfSignalsA),
  not_member(Signal,SignalsE),
  not_connected(ResetOfSignalsA,SignalsE).
not_connected([],_).
```

On the surface, it appears that **not_connected/2** is satisfactory; however, an additional stipulation exists requiring that each signal of $signals_a$ and $signals_e$ be a Prolog atom. The Prolog program **not_connected/2** does not guarantee anything is done to protect this original stipulation. This being the case, there is no guarantee that **not_connected/2** will do what it is intended to do.

At this point, there are some desired properties of a Prolog program, $\mathcal{P}$, that should be determinable. These properties include correctness and completeness with respect to the intended meaning, $\mathcal{M}$. The meaning of $\mathcal{P}$, $M(\mathcal{P})$, "is the set of ground unit goals deducible from $\mathcal{P}$" (Sterl 86:15,82-83). Thus, $\mathcal{P}$ is correct if $M(\mathcal{P}) \subset \mathcal{M}$. Further, $\mathcal{P}$ is complete if $\mathcal{M} \subset M(\mathcal{P})$. Finally, $\mathcal{P}$ is correct and complete if $\mathcal{M} = M(\mathcal{P})$.

Correctness and completeness are not the only properties of $\mathcal{P}$ that are of interest. We also wish to be able to determine whether $\mathcal{P}$ terminates. For this property, a *termination domain*

[2]A signal in VHDL has a name that corresponds to the meaning of atom in Prolog. The only difference is that Prolog is case-sensitive; whereas, VHDL is case-insensitive.

of $\mathcal{P}$ must be defined. "A *termination domain* of a program $\mathcal{P}$ is a domain $\mathcal{D}$ such that every computation of $\mathcal{P}$ on every goal in $\mathcal{D}$ terminates." (Sterl 86:83) Further, a "*domain* is a set of goals, not necessarily ground, closed under the instance relation." (Sterl 86:83) Finally, "A is an *instance* of B if there is a substitution θ such that $A = B\theta$." (Sterl 86:5)

As an illustration to what is meant by a termination domain $\mathcal{D}$, consider the Prolog program for **list/1**.

```
list([]).
list([H|T]) :- list(T).
```

The termination domain for **list/1** is represented by $\mathcal{D}_{list}$. We would like the termination domain $\mathcal{D}_{list}$ to include the goal `list(X)`. Having `list(X)` in $\mathcal{D}_{list}$ would mean that every instantiation of `X` would be in $\mathcal{D}_{list}$; however, the goal

```
?- list(X).
```

results in an infinite number of solutions.

In order to impiement **list/1**, some restriction on $\mathcal{D}_{list}$ must be adopted. This has the effect of rendering the program **list/1** fragile. Tail-recursive programs that have the same form of **list/1** require special handling. Consider the following "general" structure for a Prolog program similar to **list/1**

$$
\begin{array}{l}
c(\_P_1, \ldots, \_P_o, [\,]). \\
c(P_1, \ldots, P_o, [Head|Tail]) :- \\
\quad goal_1, \\
\quad\quad \vdots \\
\quad goal_m, \\
\quad c(SubP_1, \ldots, SubP_o, Tail).
\end{array}
$$

where o is the number of parameters passed to the program and m is the number of goals in the clause body of the program. A goal $goal_1$ may be defined to restrict the form $[Head|Tail]$ may take so as to guarantee a larger termination domain.

A restriction on the structure of the list is added to **list/1** to ensure termination over a larger $\mathcal{D}$. For this purpose we will define a meaning $\mathcal{M}_{atom\_list}$, **atom_list/1**, as a definition for the list structure to be used.

atom_list($List$) $\Leftarrow$ *Either List is empty or each element of List is an atom.*

In Prolog, the program $\mathcal{P}_{atom\_list}$ appears as

```
atom_list([]).
atom_list([Head|Tail]) :-
  atom(Head),
  atom_list(Tail).
```

The Prolog program $\mathcal{P}_{atom\_list}$ makes use of the built-in Prolog function **atom/1**. Anything meeting the requirements of **atom_list/1** will either be the empty list [] or a list whose elements are atoms, and every list comprising only of atoms will meet the requirements of **atom_list/1**. Further, **atom_list/1** will terminate for anything supplied to it as a proper Prolog parameter under the assumption that finite structures are used.

A list conforming to **atom_list/1** will either be [] or be in the form $[e_n, e_{n-1}, \ldots, e_1]$ where each e_i, $1 \leq i \leq n$, is an atom. For the case of **atom_list([])**, we find that the Prolog program $\mathcal{P}_{atom\_list}$ is consistent because it is satisfied by []. For the case of $[e_n, e_{n-1}, \ldots, e_1]$ where each e_i, $1 \leq i \leq n$, is an atom we have the following. If we assert the goal, $atom\_list([e_n, e_{n-1}, \ldots, e_1])$, we find the first clause is not satisfiable. However, the second clause is satisfiable. Performing an expansion on the second clause of **atom_list/1**

$$atom\_list([e_n|[e_{n-1},\ldots,e_1]]) :-$$
$$atom(e_n),$$
$$atom\_list([e_{n-1},\ldots,e_1]).$$

Assuming $atom\_list([e_{n-1},\ldots,e_1])$ true and knowing that $atom(e_n)$ is also true we can see that through induction, $atom\_list([e_n,e_{n-1},\ldots,e_1])$ is accepted by $\mathcal{P}_{atom\_list}$. Therefore, $\mathcal{M}_{atom\_list} \subset M(\mathcal{P}_{atom\_list})$.

To show that $\mathcal{P}_{atom\_list}$ is correct, first consider the base case of **atom_list([])**. This is acceptable to $\mathcal{P}_{atom\_list}$. Assuming that **atom_list(X)** is asserted, through unification the result is still only **atom_list([])**. Nothing else is acceptable to the first clause of **atom_list/1**. Assume now that for $[e_n,e_{n-1},\ldots,e_1]$ some e_i is not an atom and that $atom\_list([e_{n-1},\ldots,e_1])$ succeeds. This would mean that $atom(e_i)$ is true which would be a contradiction. Thus, $M(\mathcal{P}_{atom\_list}) \subset \mathcal{M}_{atom\_list}$ showing that the program $\mathcal{P}_{atom\_list}$ is correct and complete.

By adding **atom/1** to form $\mathcal{P}_{atom\_list}$ we have increased the previous termination domain $\mathcal{D}_{list}$. The termination domain $\mathcal{D}_{atom\_list}$ may now include any acceptable Prolog structure. In order to show that the program $\mathcal{P}_{atom\_list}$ terminates for $\mathcal{D}_{atom\_list}$ we first consider an empty list []. In the case of the empty list, the first clause of **atom_list/1** is satisfied. Attempting to satisfy the second clause results in failure and terminates the execution of **atom_list/1**. In the case of an acceptable list of size greater than an empty we have $[e_n,e_{n-1},\ldots,e_1]$. The second clause of **atom_list/1** is tail recursive on a list that decreases in size of one with each invocation of **atom_list/1**. The head starts with a list n elements long. It then invokes itself with a list of size $n-1$ until the empty list, [] is left which is terminated through the first clause of **atom_list/1**.

For some list failing to conform to **atom_list/1**, the case of a list $[e_n,e_{n-1},\ldots,e_1]$ is considered where some e_i is not an atom. This case may occur where e_i is a variable, list, or compound structure. For e_i, **atom_list/1** would fail at its $(n-i)+1$ invocation. Since no other clauses of **atom_list** exist that might accept e_i, the $n-i$ invocation fails. All invocations up to the $n-i$ invocation also fail and terminate.

The program $\mathcal{P}_{atom\_list}$ can be used as a type-checking routine to guarantee conformance of a list to the structure defined by **atom_list/1**. This guarantee is very important in that all programs whose termination domain is restricted to those structures defined by **atom_list/1** may be used after **atom_list/1** with the guarantee that the structure used is within their termination domain. Prolog programs, $\mathcal{P}_c$, of the form,

$$
\begin{array}{l}
c(\_P_1, \ldots, \_P_o, [\,]). \\
c(P_1, \ldots, P_o, [Head|Tail]) :- \\
\quad goal_1, \\
\quad\quad \vdots \\
\quad goal_m, \\
\quad c(SubP_1, \ldots, SubP_o, Tail).
\end{array}
$$

fall into this category provided $goal_1, \ldots, goal_m$ can be guaranteed to terminate.

The original Prolog program for **not_connected/2** can be rewritten to use **atom_list/1**.

```
not_connected(SignalsA,SignalsE) :-
  atom_list(SignalsA),
  atom_list(SignalsE),
  not_connected_sub(SignalsA,SignalsE).

not_connected_sub([],_).
not_connected_sub([Signal|SignalsA],SignalsE) :-
  not_member(Signal,SignalsA),
  not_member(Signal,SignalsE),
  not_connected_sub(SignalsA,SignalsE).
```

The program **atom_list/1** is used as a type-checking mechanism to ensure that parameters passed to **not_connected/2** are in an acceptable form. Without **atom_list/1**, **not_connected/2** would have to have its termination domain restricted. The Prolog program **not_connected_sub/2** is of the form described by the program $\mathcal{P}_c$. The termination domain $\mathcal{D}_{not\_connected\_sub}$ is guaranteed by **atom_list/1**, therefore, we can be assured that it will terminate provided **not_member/2** terminates. All that is left is to define $not\_member(Signal, SignalList)$.

The property, $not\_member(Signal, SignalList)$, may be defined in the following manner.

not_member(*Signal*, *SignalList*) $\Leftarrow$ *Signal is not a member of SignalList.*

The following is a Prolog definition for *not_member*(*Signal*, *SignalList*).

```
not_member(_,[]).
not_member(Node,[Head|Tail]) :-
  Node \== Head,
  not_member(Node,Tail).
```

The Prolog program **not_connected_sub/2** falls within the form of $\mathcal{P}_c$. When used within **not_connected_sub/2** its termination domain is also restricted by **atom_list/1**. The Prolog program **not_member/2** can be shown to be correct and complete in the same manner as **atom_list/1**.

Finally, to ensure that each component of a logic extraction rule is unique, the location information from the layout specification is used. The location information is usually in cartesian coordinates in *magic* and will be assumed so. Further, the coordinates will be integers. A collection of logic programs to ensure uniqueness is shown below.

```
coordinate_list([]).
coordinate_list([[X,Y]|RestOfCoordinates]) :-
    integer(X),
    integer(Y),
    coordinate_list(RestOfCoordinates).

unique_component([]).
unique_component([[X,Y]|Tail]) :-
        not_member([X,Y],Tail),
        unique_component(Tail).
```

The logic program **coordinate_list/1** is constructed in the same fashion as **atom_list/1**. The logic program **coordinate_list/1** provides a type-checking mechanism for **unique_component/1** thereby ensuring termination. The logic program **not_member** also provides additional service without changing its meaning.

Template for Logic Extraction

In this section, the general template for defining a logic extraction rule is presented. In keeping with the description of a structural VHDL description given earlier, the logic extraction rule will ensure the outlined properties are kept. However, the procedural aspects of Prolog must also be considered so that extraction may be performed in an automated fashion. Accordingly, six procedural steps are performed in the order indicated.

1. Identify the component from its lower-level components.
2. Ensure that the identified lower-level components are unique.
3. Check that the values of internal nodes do not match other nodes.
4. Delete the lower-level components from the component netlist.
5. Add the newly found component to the component netlist.
6. Check to see if there are more lower-level components.

Step 1 must occur first, step two must occur second, step three must occur third, and step 6 must occur last; however, the order of steps 4 and 5 is not important.

The general template for forming a Prolog extraction rule is:

head :-
  matching_goal$_1$,
    .
    .
    .
  matching_goal$_n$,
  coordinate_list([[X_1, Y_1], ..., [X_n, Y_n]]),
  unique_component([[X_1, Y_1], ..., [X_n, Y_n]]),
  not_connected(*Signals*$_a$, *Signals*$_e$),
  retract_goal$_1$,
    .
    .
    .
  retract_goal$_n$,
  find_anomaly_list(*head*(*argument_list*),*Signals*$_a$),
  asserta(*head*(*argument_list*)),
  fail.
head.

Each of the coordinates used by **coordinate_list/1** and **unique_component/1** are derived from each of the *matching_goal*$_i$, where $1 \leq i \leq n$. A discussion of **find_anomaly_list/2** is provided later in this chapter.

Currently, the logic extraction process is divided into two parts. The first part, called Level-1, is a collection of rules for translating transistors of a particular technology to logical components. The second part, called Level-N, is automatically generated (Dukes 91b) from VHDL descriptions using *vhdl_parser* (Reint 90) to translate VHDL descriptions into Prolog logic extraction rules. A Level-1 rule-set for CMOS and a Level-1 rule-set for the Vitesse GaAs process have been developed. The Level-1 rule-set for the Vitesse GaAs process has not been extensively tested. The CMOS Level-1 rule-set contains rules for extracting inverters, transmission gates, clocked inverters, N-input **NAND** gates, N-input **NOR** gates, and three types of **EXCLUSIVE-OR/EXCLUSIVE-NOR** gates. An additional rule for D-latches is also included.

Definitions for Using CMOS Level-1 Rules

The level-1 rules have to be generated by hand, though it is possible (but not desirable) to use *vhdl2ges* (Dukes 91b) to generate level-1 rules from a structural VHDL description using

transistor-type components. Since the drain and source of a CMOS transistor are interchangeable in their abstract view, some additional definitions are adopted to accommodate the abstract view. In this section, an explanation and examples of how Prolog handles abstraction are presented. For the representation of transistor-facts, we adopt the following definitions.

Definition 1 $p(G, D, S, W, L, X, Y)$ is a predicate that describes a p-type MOS transistor with a gate G, drain D, source S, channel width W, channel length L, x-location X, and y-location Y.

Definition 2 $n(G, D, S, W, L, X, Y)$ is a predicate that describes an n-type MOS transistor with a gate G, drain D, source S, channel width W, channel length L, x-location X, and y-location Y.

The arity (number of arguments) of seven for both the p and n predicates assumes that the physical bulk (or substrate) connection of the MOS transistor is biased correctly. Furthermore, the description adopted is for p-type and n-type enhancement mode MOS transistors. Typical p-type and n-type MOS transistors in Prolog appear as

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

In the physical sense, the actual drain and source are determined by the biasing of the device. In the abstract sense (i.e., *magic* and *extract*), the drain and source are freely interchanged. Implementations of the p-type and n-type transistors in MOS layout freely interchange the drain and source (Weste 85). In order to express this abstract aspect and to suppress information concerning length and width, some further definitions are adopted.

Definition 3 The predicate 'ptrans' is defined in terms of the predicate 'p' by the implication

$$\forall G, D, S, X, Y\ [(p(G, D, S, \_, \_, X, Y) \vee p(G, S, D, \_, \_, X, Y)) \Rightarrow \mathit{ptrans}(G, D, S, X, Y)].$$

Definition 4 The predicate 'ntrans' is defined in terms of the predicate 'n' by the implication

$$\forall G, D, S, X, Y\ [(n(G, D, S, \_, \_, X, Y) \vee n(G, S, D, \_, \_, X, Y)) \Rightarrow \mathit{ntrans}(G, D, S, X, Y)].$$

The underscore indicates unnecessary information. The Prolog rules to describe Definitions 3 and 4 are

```
ptrans(G,D,S,X,Y) :-
  p(G,D,S,_,_,X,Y).
ptrans(G,D,S,X,Y) :-
  p(G,S,D,_,_,X,Y).

ntrans(G,D,S,X,Y) :-
  n(G,D,S,_,_,X,Y).
ntrans(G,D,S,X,Y) :-
  n(G,S,D,_,_,X,Y).
```

Assuming that **ptrans/5** and **ntrans/5** exist in a file called **trans.pro** on a UNIX system and that Quintus Prolog is also installed on the same system, **ptrans/5** and **ntrans/5** may be loaded into Prolog in the following manner.

```
% prolog

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile['trans.pro'].
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 0.267 sec 564 bytes]

yes
| ?-
```

The system prompt is **%**. The Prolog prompt is **| ?-** . The file, **trans.pro**, was loaded into Prolog using the Quintus Prolog procedure called **compile/1**. The Prolog function, **compile/1**, compiles the contents of the file, **trans.pro**, into the current Prolog session.

Assume the following transistor netlist exists in a UNIX file called intrans.pro.

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

The transistor information would be read into Prolog in the following manner.

```
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.100 sec 564 bytes]

yes
| ?-
```

All of the p-type transistors may be listed by querying Prolog in the following manner.

```
| ?- p(G,D,S,W,L,X,Y).

G = nINPUT,
D = nvdd,
S = nOUTPUT,
W = 3,
L = 6,
X = 1254,
Y = 387 ;

G = nADDIN,
D = nA_INPUT,
S = nA_SELECT,
W = 3,
L = 6,
X = 39887,
Y = -3091 ;

no
| ?- halt.

[ End of Prolog execution ]
%
```

The upper case letters, G, D, S, W, L, X, and Y, designate variables to be satisfied by Prolog. The ; (semicolon) is used to request further information from the transistor database that satisfies the request. Otherwise, a carriage return not preceded by a ; would have terminated the search. The `halt/0` predicate tells Prolog to terminate and return to the system prompt.

Assume now that the transistor netlist consists of the following components.

```
p(nINPUT,nvdd,nOUTPUT,3,6,1254,387).
p(nINSTATE,nNOTINSTATE,nvdd,3,6,1254,387).
p(nADDIN,nA_INPUT,nA_SELECT,3,6,39887,-3091).
n(nINPUT,ngnd,nOUTPUT,3,6,1260,387).
```

Assume also that we are interested in finding transistors with nvdd on either the drain or source of a p-type transistor. The objective may be accomplished in one of two ways. The first method is to express two queries to Prolog using

```
| ?- p(G,nvdd,S,W,L,X,Y) ; p(G,D,nvdd,W,L,X,Y).
```

In the above example, the ; is used to logically **OR** the two queries. The result of the two queries is displayed below.

```
% prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.167 sec 880 bytes]

yes
| ?- p(G,nvdd,S,W,L,X,Y) ; p(G,D,nvdd,W,L,X,Y).

G = nINPUT,
S = nOUTPUT,
W = 3,
L = 6,
X = 1254,
Y = 387,
D = _149 ;

G = nINSTATE,
S = _55,
W = 3,
L = 6,
X = 1254,
Y = 387,
D = nNOTINSTATE ;

no
| ?-
```

However, the Prolog rules stated in **ptrans/5** will accomplish the same task, as shown in the following:

```
% prolog

Quintus Prolog Release 2.2 (Sun-3, Unix 3.2)
Copyright (C) 1987, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- compile(['trans.pro']).
[compiling /people/dukes/class/trans.pro...]
[trans.pro compiled 0.250 sec 564 bytes]

yes
| ?- ['intrans.pro'].
[consulting /people/dukes/class/intrans.pro...]
[intrans.pro consulted 0.100 sec 688 bytes]

yes
| ?- ptrans(G,nvdd,S,X,Y).

G = input,
S = output,
X = 1254,
Y = 387 ;

G = instate,
S = notinstate,
X = 1254,
Y = 387 ;

no
| ?-
```

This example demonstrates that rules using transistors may not be concerned with the interchangeability of the drain and source. Assuming that circuit function, not timing, is of primary interest, unnecessary information (e.g., gate width and length) may be easily dropped when performing extraction.

Guaranteeing Termination for Logic Extraction

For the purpose of this discussion, the following representation will be used.

- $LE(Case)$ will represent the $Case$ under consideration for logic extraction.
- $P_{terminate}$ will represent the property of termination being true.

Further, there are only a finite number of facts in the facts data base.

Several cases will be considered. In all cases, m and n will represent natural numbers excluding zero. The first case addresses the possibility of replacement of m components by n where $m < n$. The second case addresses the replacement of m components by n components where $m = n$. Finally, the last case addresses the replacement of m components by n components where $m > n$. For the discussion of logic extraction, **not_connected/2** and **find_anomaly/2** are assumed to terminate.

Case 1. The first case to discuss deals with the replacement of m components by n components where $m < n$. What we want to show is that

$$LE(m < n) \Rightarrow P_{terminate}.$$

Logic extraction rules constructed through GES only assert a single component. Thus, $n = 1$; however, $n > m$ by our original assumption and $m \geq 1$ meaning that n can never be 1. Therefore, $LE(m < n)$ is false and the assertion for case 1 is trivially proven.

Case 2. The second case to discuss concerns the replacement of m components by n components where $m = n$ or

$$LE(m = n) \Rightarrow P_{terminate}.$$

There are two subcases to be proven. These cases are[3]

$$((m > 1) \wedge (n > 1)) \vdash LE(m = n) \Rightarrow P_{terminate}$$

[3] A discussion on the form $\Gamma \vdash \alpha$ is in Appendix E.

and

$$((m = 1) \wedge (n = 1)) \vdash LE(m = n) \Rightarrow P_{terminate}.$$

Rewriting the first subcase, we have

$$((m > 1) \wedge (n > 1)) \vdash ((m > 1) \wedge (n > 1)) \Rightarrow (LE(m = n) \Rightarrow P_{terminate}).$$

From Case 1 we know that n has to be 1. Therefore, $(n > 1)$ is false and this subcase is true.

For the second subcase of

$$((m = 1) \wedge (n = 1)) \vdash LE(m = n) \Rightarrow P_{terminate}$$

the value for m is valid as well as n. Thus, the extraction rule must be examined. The following is a representation of a logic extraction for the case where the component being matched is the same as the component being asserted on the component data base.

```
c :-
   c(P1, ..., Po),
   not_connected([ ],[P1, ..., Po]),
   retract(c(P1, ..., Po)),
   find_anomaly_list(c(P1, ..., Po),[]),
   asserta(c(P1, ..., Po)),
   fail.
c.
```

Both **not_connected/2** and **find_anomaly_list/2** terminate immediately due to []. The remainder requires an understanding of fact matching and **asserta/1**.

In Prolog, matching of facts starts at the head of a facts data base. Upon each retry, the succeeding fact is looked up. Prolog does not return to a preceding fact. In this fashion, a data base of facts has a head, a tail, and a progression from head to tail. The Prolog procedure **asserta/1** places a fact before the head of a fact data base. By modifying the facts data base in this fashion,

a newly asserted fact will not cause infinite backtracking through the facts data base since it is considered a preceding fact. Furthermore, the facts data base is assumed to be finite. Therefore, by using **asserta/1** we are guaranteed termination.

For the case where the matching goal and the fact being asserted are not the same, the process is trivial. Each time a goal is matched, a new component of a different name is asserted. This process continues until all facts are retracted. The rule then terminates.

Case 3. The last case to discuss concerns the replacement of m components by n components where $m > n$ or

$$LE(m > n) \Rightarrow P_{terminate}.$$

From case 1, we know that n must be 1. Since $m > 1$ by substitution, we see that the number of components is decreasing rather than ever increasing. At worst, the logic extraction process will continue until only one component is left.

Design Rule Checking

The extraction methodology previously described has only been concerned with extracting normal circuits. However, we cannot assume that the circuit to be extracted is free from design errors. Since errors may exist in a design, we must be prepared to find them. These errors occur because the external interconnections of a component are configured in an inconsistent condition. In this section, the problem of identifying these errors will be discussed.

Identifying External Design Errors The CMOS designs described earlier were based on a design style of a particular designer or group of designers. Just as CMOS designs are based on a design style, so too are design errors. A few of the types of design errors possible are shown in Figure 9. It is important to point out that the following circuits are considered to be errors because they are not normally used in the design of a VLSI circuit.

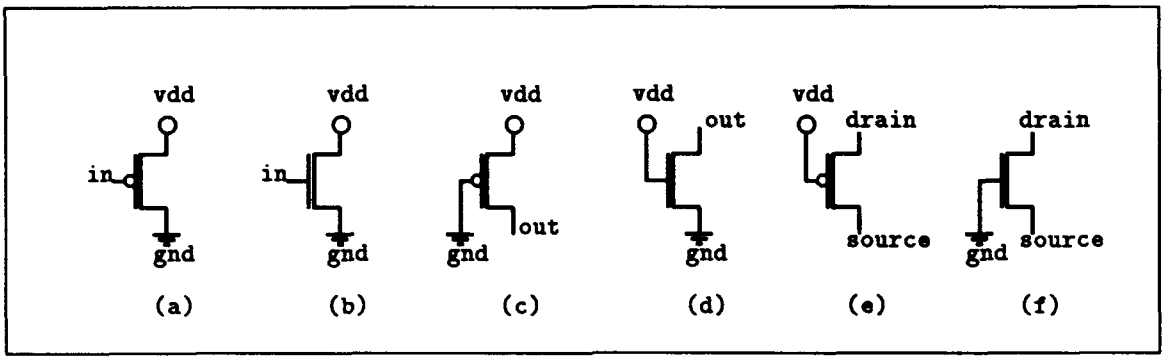


Figure 9. Some Transistor-Level Design Errors.

Subfigures a and b of Figure 9 typify a dangerous circuit. This type of design error may result from one of several actions on the part of the layout system used. If plowing or some other form of circuit rearrangement is being performed, it is possible to connect the terminals of the transistor in the fashion shown. During layout in *magic*, routing over subcells with metal layers that accidentally contact the same layer of lower subcells may also cause this problem. In either case, the result is a circuit that, when turned on, will cause a short-circuit between Vdd and GND.

The physical nature of such a circuit is not the only concern in Subfigures a and b of Figure 9. Assuming for discussion that we were only concerned with showing

$$Implementation \Rightarrow Specification$$

the result of Subfigures a and b could lead to an invalid conclusion. Essentially, the resulting equation from such an error might be

$$FALSE \Rightarrow Specification.$$

The specific problem of connecting Vdd and GND at the transistor level in HOL has been raised by (Gupta 91:20). The problem is generally referred to as "false implies everything" (Camil 86:22). A

false antecedent can lead the naive verification method to conclude an implementation meets a specification. Therefore, checking for such errors can reduce occurrences where improper conclusions about hardware might be reached.

The circuits in subfigures c and d of Figure 9 are a little less destructive than the circuits discussed earlier. However, they may indicate design errors. These circuits may be easily replaced by a metal line connected to Vdd for subfigure c or GND for subfigure d. Even though these circuits may be caused by the problems indicated for subfigures a and b, they may also be the result of tying inputs to arrays of standard cells high or low. Subfigures e and f of Figure 9 demonstrate another possible design error. As with the circuits in subfigures c and d, their creation may be accidental or incidental. The following "error" definition is offered.

Definition E1 $\forall In, Out, Drain, Source,$

1. ptrans(In,vdd,gnd) is an error;
2. ntrans(In,vdd,gnd) is an error;
3. ptrans(gnd,vdd,Out) is an error;
4. ntrans(vdd,Out,gnd) is an error;
5. ptrans(vdd,Drain,Source) is an error;
6. ntrans(gnd,Drain,Source) is an error.

Recognition of design flaws is not limited to single transistors. Erroneous designs consisting of groups of transistors may also occur. Figures 10 provides examples of designs that may be considered design errors. For these structures, another "error" definition may be considered.

Definition E2 $\forall Pg, Ng, In, Out,$

1. invZ(Pg,Pg,In,Out) is an error;
2. invZ(Pg,Ng,In,In) is an error;
3. tgate(Pg,Pg,In,Out) is an error;
4. tgate(Pg,Ng,In,In) is an error;
5. inv(In,In) is an error.

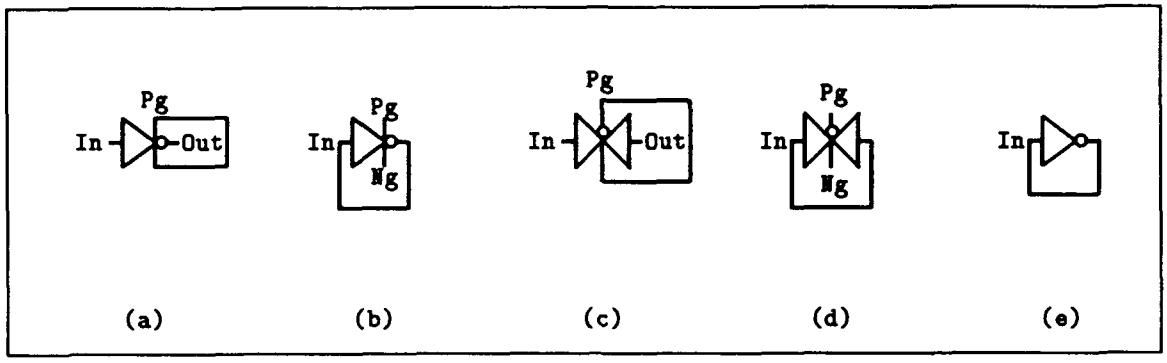


Figure 10. Some Gate-Level Design Errors.

The list of errors included in Definitions E1 and E2 is not complete. For some designs, some of the enumerated errors may not be errors at all. Therefore, definitions for design errors are declared within the domain of the design style under consideration.

Prolog Implementation for Identifying External Design Errors This section describes two methods for using Prolog to identify design errors. The first method described is an interactive one where the user provides statements to be satisfied by Prolog from the component database. The second method described allows the user to specify a list of Prolog rules that may be stored in a file and executed at a later time.

Definitions E1 and E2 designate certain component configurations to be erroneous. Using Prolog interactively, these errors may be identified easily. The following is a demonstration of how the component database is examined for the occurrence of the first type of error in Definition E1.

```
| ?- ptrans(In,nvdd,ngnd,X,Y).
In = n20_024_14A_BAR,
X = 5722,
Y = 141 ;
In = n20_024_13A_BAR,
X = 5722,
Y = 506 ;
In = n20_024_12A_BAR,
X = 5722,
Y = 798 ;
In = n20_024_11A_BAR,
X = 5722,
Y = 1017
```

Notice the use of the two additional fields, X and Y. These fields contain location information that may be used to find the errant components. Through the use of Definition 3, we were able to ask Prolog to identify those transistors that satisfied one of the design error types in Definition E1. We may also perform the same query for higher level components as shown below.

```
| ?- tgate(Pg,Ng,In,In,X,Y).
no
| ?- tgate(Pg,Pg,In,Out,X,Y).
Pg = n12_0typeIIId_4XOR,
In = n12_0typeIIId_4COUT1,
Out = n12_0typeIIIb_15COUT1,
X = -306,
Y = 97 ;
Pg = n12_0typeIIIb_15XOR,
In = n12_0typeIIIb_15COUT1,
Out = n12_0typeIIIa_14COUT1,
X = -306,
Y = 170 ;
Pg = n12_0typeIIIc_9XOR,
In = n12_0typeIIIc_9COUT1,
Out = n12_0typeIIIb_14COUT1,
X = -290,
Y = 316
```

In the previous example, the component database was queried for the existence of any transmission gates that met condition 4 of Definition E2. In this case, no transmission gates were found. However, when Prolog was queried for the existence of transmission gates that violated condition 3 of Definition E2, several instances were found and reported.

Design errors may also be found through the establishment of Prolog rules prior to performing duplicate transistor reduction and extraction in the case of Definition E1. Furthermore, the extraction process may be performed after extraction using Level-1 rules to identify components that satisfy to Definition E2. The following is an example of a rule used to find a design error identified in Definition E1.

```
/*  Error type 1   */
find_error :-
  ptrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ptrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_p(G,nvdd,ngnd),
  fail.
/*  Error type 2   */
find_error :-
  ntrans(G,nvdd,ngnd,X,Y),
  write('Bad trans, '),write(ntrans(G,nvdd,ngnd,X,Y)),
  write(': removed'),nl,
  remove_n(G,nvdd,ngnd),
  fail.
/*  Error type 3   */
find_error :-
  ptrans(ngnd,nvdd,S,X,Y),
  write('Straight wire, '),write(ptrans(ngnd,nvdd,S,X,Y)),
  write(': removed'),nl,
  remove_p(ngnd,nvdd,S),
  fail.
/*  Error type 4   */
find_error :-
  ntrans(nvdd,ngnd,S,X,Y),
  write('Straight wire, '),write(ntrans(nvdd,ngnd,S,X,Y)),
  write(': removed'),nl,
  remove_n(nvdd,ngnd,S),
  fail.
/*  Error type 5   */
find_error :-
  ptrans(nvdd,A,B,X,Y),
  write('Open connection, '),write(ptrans(nvdd,A,B,X,Y)),
  write(': removed'),nl,
  remove_p(nvdd,A,B),
  fail.
/*  Error type 6   */
find_error :-
  ntrans(ngnd,A,B,X,Y),
  write('Open connection, '),write(ntrans(ngnd,A,B,X,Y)),
  write(': removed'),nl,
  remove_n(ngnd,A,B),
```

```
  fail.
find_error.
```

Notice that `find_error.` is listed last. This is to provide a successful outcome when all of the previous clauses fail.

The following Prolog rules are used to identify design errors that conform to Definition E2.

```
find_more_errors :-
  clk_inv(P,P,In,Out,X,Y),
  write('Screwy clk_inv, '),write(clk_inv(P,P,In,Out,X,Y)),
  write(': removed'),nl,
  retract(clk_inv(P,P,In,Out,X,Y)),
  fail.
find_more_errors :-
  clk_inv(Pg,Ng,Bad,Bad,X,Y),
  write('Oscillating clk_inv, '),write(clk_inv(Pg,Ng,Bad,Bad,X,Y)),
  write(': removed'),nl,
  retract(clk_inv(Pg,Ng,Bad,Bad,X,Y)),
  fail.
find_more_errors :-
  tgate(P,P,In,Out,X,Y),
  write('Screwy tgate, '),write(tgate(P,P,In,Out,X,Y)),
  write(': removed'),nl,
  retract(tgate(P,P,In,Out,X,Y)),
  fail.
find_more_errors :-
  tgate(Pg,Ng,Bad,Bad,X,Y),
  write('Worthless tgate, '),write(tgate(Pg,Ng,Bad,Bad,X,Y)),
  write(': removed'),nl,
  retract(tgate(Pg,Ng,Bad,Bad,X,Y)),
  fail.
find_more_errors :-
  inv(Bad,Bad,X,Y),
  write('Oscillating inv, '),write(inv(Bad,Bad,X,Y)),
  write(': removed'),nl,
  retract(inv(Bad,Bad,X,Y)),
  fail.
find_more_errors.
```

Identifying Internal Design Errors The Prolog rule, **find_anomaly_list/2** is used to identify "global"[4] connectivity errors such as the one shown in Figure 7. Identifying this class of error is important, since connections that violate the component boundary implied by a structural

[4]The discussion of global and local connectivity is in Chapter 3.

VHDL description change the behavior of the component to be extracted. The following Prolog program is a definition for identifying this problem.

```
find_anomaly_list(_,[]).
find_anomaly_list(Comp,[Node|Rest]) :-
  find_anomaly(Comp,Node),
  find_anomaly_list(Comp,Rest).
```

The Prolog rule calls upon a series of clauses under **find_anomaly/2** that compares one of the internal nodes of the component being extracted to the external connections of the all other components in the component database. An example of how **find_anomaly/2** is defined to examine transistors and a transmission gate is the following.

```
find_anomaly(Comp,Node) :-
  (ptrans(Node,_,_,X,Y);
   ptrans(_,Node,_,X,Y);
   ntrans(Node,_,_,X,Y);
   ntrans(_,Node,_,X,Y)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a transistor at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
find_anomaly(Comp,Node) :-
  (tgate(Node,_,_,_,X,Y);
   tgate(_,Node,_,_,X,Y);
   tgate(_,_,Node,_,X,Y);
   tgate(_,_,_,Node,X,Y)),
  write('Failure extracting component '),write(Comp),
  write('.'),nl,write('     Internal node, '),write(Node),
  write(', connected to a tgate at X:'),
  write(X),write(', Y:'),write(Y),write('.'),nl.
find_anomaly(_,_).
```

The Prolog program **find_anomaly/2** is set up to produce warning messages to the designer and continue hunting for other possible connectivity problems. Further, **find_anomaly/2** is set to succeed in this case so that errors for other components may be found. Use of a `!,fail` could have been used to force failure upon the first encounter if desired.

A separate **find_anomaly/2** is generated for each type of component that can exist in the component data base. Additional **find_anomaly/2** clauses are generated automatically by *vhdl2ges* (Dukes 91b) for each component that can be generated on the component data base by an extraction rule.

The Prolog program **find_anomaly_list/2** corresponds to the form of $\mathcal{P}_c$. The list passed to **find_anomaly_list/2** is guaranteed to be in the termination domain $\mathcal{D}_c$ by **atom_list/1** used in **not_connected/2**. Further, **find_anomaly/2** executes only while component-facts of the type being searched exist in the component database. Thus, we can be reasonably certain that **find_anomaly_list/2** will always terminate.

V. Delay Models for VHDL

Timing information used for hardware design may be viewed from several perspectives. The physical representation of the design (e.g. mask layout description) can provide close approximations of the actual resistive and capacitive loading encountered in a design. At a more abstract level of the hardware design, a VHDL model may be used to predict hardware timing from a direct or indirect perspective. The discussion that follows presents three methods of accomplishing pin-to-pin critical path analysis. The type of critical path analysis examined in this research performs the process of extracting pin-to-pin critical paths in two steps. A calculation of propagation delays through the lowest-level components is performed in the first step. The second step involves summing delays from inputs to outputs for all possible critical paths.

For instance, the physical layout description may be viewed as the lowest level perspective in critical path analysis. Extracting pin-to-pin critical paths from a layout description using logic extraction occurs in the following manner.

1. Propagation values are calculated from known capacitive and resistive loading in the circuit.
2. Delays are summed along paths from input pins to output pins.

Calculating Delays from Layout

Shown in Figure 11 is a simple model of a CMOS inverter used to calculate delay[1]. The figure represents the resistive and capacitive elements that would be encountered. The elements in the circuit are R_p for the resistance through the channel of the p-type MOS transistor, R_n for the resistance through the channel of the n-type MOS transistor, R_L for the "lumped[2]" resistance of the output node, C_L for the "lumped[3]" capacitance of the output node, $C_1 \ldots C_n$ for the capacitance[4]

[1]This model was chosen to demonstrate one method for incorporating delay calculations into the extraction process of GES.

[2]the resistance as computed on a node by *magic*'s extract

[3]the sum of the capacitances between the output node and all other nodes reported by *ext2sim*

[4]Since *ext2sim* does not produce gate capacitances, a gate capacitance is also computed from the existing transistors in the transistor database in GES.

on every MOS gate connected to the output node, and V_O for the voltage between the output node and GND.

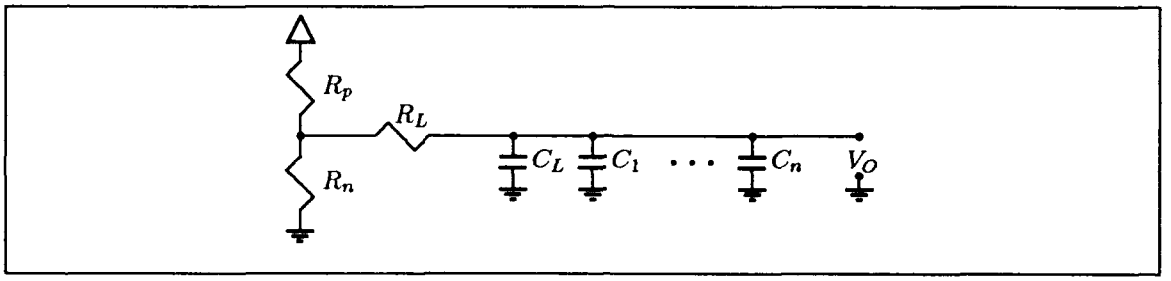


Figure 11. A Simple Delay Model.

From Figure 11, two delay models may be constructed. The general equation used for propagation delay is[5]

$$\tau = \sum R \ \sum C. \tag{7}$$

For the case where the p-type MOS transistor of the circuit is turned on, the n-type MOS transistor of the circuit is off, and V_O is equal to ground, the propagation delay may be described by

$$\tau = (R_p + R_L)(C_L + \sum_{i=1}^{n} C_i). \tag{8}$$

Likewise, for the case where the n-type MOS transistor of the circuit is turned on, the p-type MOS transistor of the circuit is off, and V_O is equal to Vdd, the propagation delay may be described by

$$\tau = (R_n + R_L)(C_L + \sum_{i=1}^{n} C_i). \tag{9}$$

If we consider the CMOS inverter as a degenerate case of the **NOR**-gate and the **NAND**-gate, propagation delays may also be calculated by the appropriate parallel and series formulas for the p-type and n-type MOS transistor networks of both types of gates. Calculating propagation

[5]This model for computing delays is typical of one method shown in the literature (Ouste 84)

delays based on this model appears valid, provided the output node terminates only on gates of MOS transistors. Should pass transistor logic exist in the circuit, a different view must be adopted.

For pass-transistor logic, the following practice is adopted. A pass-transistor path is one which follows along the drain to source of a p-type or n-type MOS transistor without a termination to Vdd or GND. The total load resistance, R_L, is determined from the appropriate parallel and series computations of resistances along all pass-transistor paths connected to the output node. Furthermore, all capacitances along all pass-transistor paths connected to the output node are summed. Essentially, a worst-case calculation of possible loading is performed by assuming all pass transistors are in a conducting state. A worst-case calculation is necessary for conducting critical path analysis.

Determining Propagation Delay in VHDL

There are three methods that may be used to determine propagation delay from a VHDL model. The first method uses the timing information explicitly stated in the VHDL model through the signal assignment statement. The second method calculates the timing information from the loading on a particular input. The third method combines the delay based on loading and the explicitly stated delay in the VHDL model.

Delay Model Specified in VHDL As indicated in Figure 12, delays are specified by the VHDL model for all components regardless of the output drive. In this case, a specified delay on a component in VHDL implies the drive required by that physical component to meet that delay. Propagation delays along paths are then determined in a way similar to step two for the physical layout description.

Delay Model for Loading in VHDL Shown in Figure 13 is a model for calculating delay of a component output. The function

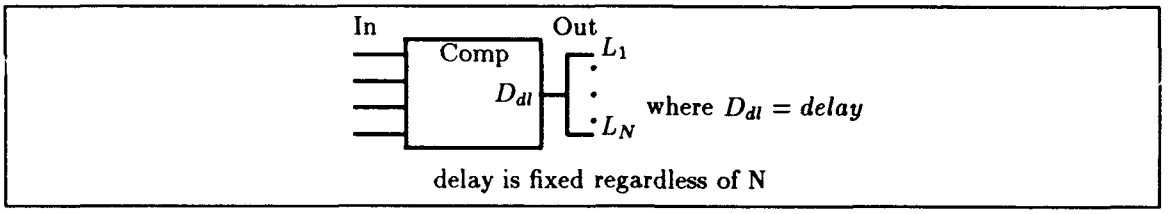


Figure 12. Delays Specified in Description.

$$delay_{dr} = f([L_1, ..., L_N], D_{dr}) \tag{10}$$

returns a delay value based on the loading of other components being driven by the output and an internal drive capacity, D. Once the delays are calculated for all components, delays are summed along paths from input pins to output pins in a fashion similar to the second step for critical path analysis of physical layout.

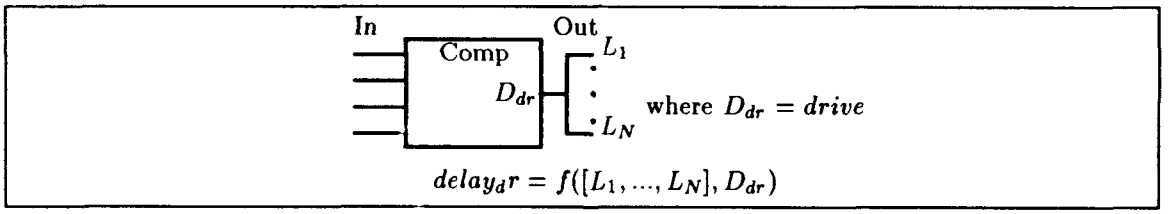


Figure 13. Delays Calculated from Fanout.

It is not obvious how the fanout for a signal may be calculated. The information may be gathered by flattening a VHDL model using a Prolog routine called *flatten*. The Prolog routine converts a hierarchical VHDL model to a gate-level component netlist. Propagation delays may then be calculated by determining the number of inputs being driven by a signal. The search is easy to perform and each component has a propagation delay calculated through this method. At this point, logic extraction may be conducted through a customized GES routine, performing pin-to-pin critical path analysis.

Hybrid Delay Model in VHDL The hybrid delay model combines the two previous models. A hybrid delay equation may be constructed as follows.

$$delay_{hy} = delay_{dr} + D_{dl} \tag{11}$$

Like the delay model for loading in VHDL, delay values for all components must be calculated first, assigning each a $delay_{dr}$. Then a $delay_{hy}$ is calculated for each component based on Eq 11. Afterwards, propagation delays are determined in a manner similar to step two for the physical layout description.

VI. Critical Path Analysis

Knowing that a layout specification matches a structural specification component-by-component and connection-by-connection is not sufficient to guarantee equivalence between both structural specification and a layout specification. Other properties, e.g., power requirements, circuit delays, and circuit reliability, are important to consider in addition to circuit function. If a circuit does not meet a timing constraint specified in its structural specification, then it is useless regardless if it is funtionally equivalent to its structural specification.

Presented in this chapter is one approach to performing pin-to-pin critical path analysis of a circuit within the logic extraction process. We will show how logic extraction limits the circuit size under consideration for pin-to-pin critical path analysis and prunes many noncritical paths early. Through the process of pruning, pin-to-pin critical path analysis of very large circuits may be possible.

Though this chapter focuses on pin-to-pin critical path analysis, other properties of a circuit may be examined. We hope that the discussion on pin-to-pin critical path analysis here will provide insight as to how other properties may be extracted within logic extraction.

Consideration of Feedback in Critical Path Analysis

The contribution of feedback loops to pin-to-pin critical analysis is considered here before presenting the definitions for structures used in the pin-to-pin critical path analysis. Figure 14 is a Huffman model (Hayes 88:108-109) for a typical circuit. **CN** represents the combinational circuit network and **L** represents the latching or memory circuit. The output of the circuit depends upon the inputs and the present state values of **L**. In a stable circuit, the delays from input to output in the **CN** do not include the feedback paths. Thus, the delay of the combinational circuit is actually within the **CN** portion of the model. The asynchronous cycle time may be computed by "breaking"

the feedback paths in the VHDL model and extracting the circuit with a new VHDL model, thus adding the **CN** and **L** critical paths together.

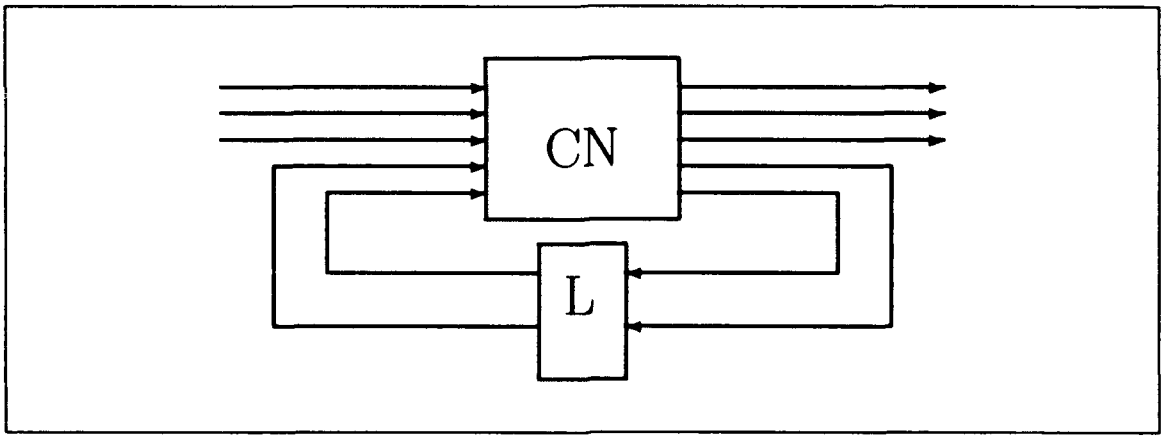


Figure 14. Huffman Model.

Extracting Critical Paths

Every component above the transistor level in GES contains a set[1] of paths. A *path*, $\mathcal{P}$, is a pair, $[L, D]$, where L[2] is an edge-bounded acyclic path (EAP) and D is the propagation delay of L. L is an ordered collection of nodes where the *head* is the first node in the EAP and the *last* is the last node in the EAP. All of the nodes in L describe a path through a circuit. For the purpose of constructing pin-to-pin critical paths through a circuit, E is a set of input and output nodes for the component being extracted, which is a subset of all the nodes in the circuit.

Path Generation Without Feedback

The following are definitions for terms and functions used to form the extraction of critical paths during the extraction process in GES. For notation, a preceding lower case letter on a label

[1] Sets and lists in the context of this discussion have different meanings, though both are represented through lists in Prolog. In the case of a set, the conventional meaning as an unordered collection of unique objects prevails. A list is considered to be an ordered collection of objects.

[2] L is used here since the structure conforms to **list/1**.

designates an atom, a preceding upper case letter on a label designates a variable, and a preceding underscore[3] designates a "don't care."

Definition 1 *A* **node** *is an interconnection label in a circuit. For notation, a node is represented by* n.

Definition 2 *An* **EAP** *is an ordered sequence of nodes. An* **EAP** *is constructed as* $[n_1, \ldots, n_m]$, *where* $m \geq 2$ *and* $n_1 \neq n_m$.

Using the symbol | as a list constructor, an EAP may be constructed as $[n_1|[n_2|[n_3|\ldots[n_m|[\,]]\ldots]]]$. The node n_1 is called the *head*, the list of nodes following n_1 is called the *tail*, and n_m is called the *last*. For notational convenience, the head of L may be referred to as head(L), the tail referred to as tail(L), and the last referred to as last(L). Functionally, the head, tail, and last may be represented as the following.

$$
\begin{aligned}
head([n_1, \ldots, n_m]) &= n_1 \\
tail([n_1, n_2, \ldots, n_m]) &= [n_2, \ldots, n_m] \\
last([n_1, \ldots, n_m]) &= n_m
\end{aligned}
$$

Definition 3 *The predicate, member(n, L), is true when* $n \in L$ *and false otherwise.*

$$
\begin{aligned}
member(N, [N|\_L]) &\rightarrow true \\
member(N, [\_N|L]) &\rightarrow member(N, L)
\end{aligned}
$$

In Prolog, member may be written as the following.

[3] In Prolog, an underscore designates a "don't care." In order to preserve the meaning of a variable location and avoid singleton variable warnings in Quintus Prolog (Quint 88), this notation was adopted.

```
member(M,[M|_L]).

member(M,[_M|L]) :- member(M,L).
```

Definition 4 *The predicate, append(L_1,L_2,L_3) is true when* $L_1 = [n_1, \ldots, n_m]$, $L_2 = [n_{m+1}, \ldots, n_{m+p}]$, *and* $L_3 = [n_1, \ldots, n_m, n_{m+1}, \ldots, n_{m+p}]$.

In Prolog, the append function may be written as follows.

```
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).

append([],L,L).
```

Let $L_1, L_2, \ldots, L_i$ be EAPs, $H_i = head(L_i), T_i = last(L_i)$ and E a set of input and output nodes for the component being extracted which is a subset of all the nodes in the circuit.

Definition 5 *join(L_1, L_2, L_3) is true* **iff**

1. *member(H_1, E),*
2. $T_1 = H_2$,
3. *member(T_1, E) is false,*
4. *member(T_2, L_1) is false,*
5. $R_2 = tail(L_2)$,
6. *and append(L_1, R_2, L_3).*

In Prolog, the join function[4] may be written as the following[5]

[4] Difference lists (Bratk 86:192) may be used to increase the efficiency of the operations shown.

[5] In some Prolog implementations (Quint 88), the `not/1` function does not exist. In this context, the `not/1` function is assumed to be defined as

```
not(Goal) :- call(Goal),!,fail.
not(_).
```

```
join([H1|R1],[T1|R2],E,L3) :-
  member(H1,E), last(R1,T1), last(R2,T2), not(member(T1,E)),
  not(member(T2,[H1|R1])), append([H1|R1],R2,L3).
```

For convenience, the symbol "$\star$" will be used to denote the join operation in the following manner.

$$[L_3 = L_1 \star L_2] \stackrel{\text{def}}{=} [join(L_1, L_2, L_3)]. \tag{12}$$

EAPs may only be constructed using the *join* predicate. In essence, the join operation may be thought of as an EAP extension function. This being the case, we may show the following.

Lemma 1 *An EAP, L, with $head(L) \notin E$, will have only two nodes ($m = 2$).*

Proof

By Definition 5, only EAPs with $head(L) \in E$ may be extended. Therefore, all EAPs with $m > 2$ must have their $head(L) \in E$ leaving lists with $head(L) \notin E$ with only $m = 2$.

Theorem 1 *Let L be an m node EAP where $2 \leq m$ and $1 \leq i \leq m$, $1 \leq j \leq m$, and $i \neq j$. $\forall n \in L$, if $n_i = n_j$ then L is not a valid EAP.*

Proof

base case: Assume $m = 2$. Then $L = [n_1, n_2]$. Assume $n_1 = n_2$. This is not consistent with Definition 2. Therefore, $n_1 \neq n_2$.

hypothesis: Assume $m > 2$. Then $L = [n_1, \ldots, n_m]$. $n_1 \neq n_m$ by Definition 2. By Lemma 1, we know that $n_1 \in E$. Furthermore, we assume that L contains no feedback loops.

induction: $L_{m+1} = [n_1, \ldots, n_{m+1}]$ and $L_{m,m+1} = [n_m, n_{m+1}]$. We need to show that

$$L_{m+1} = L_m \star L_{m,m+1}.$$

The EAP L_{m+1} cannot exist or be formed if $n_m \in E$ by Definition 5. Thus $n_m \notin E$ and by Lemma 1, the EAP, $L_{m,m+1}$, must have only two nodes. For the operation to be valid, $n_{m+1} \notin L_m$ by Definition 5 thereby avoiding a feedback loop.

From Theorem 1, it is evident that EAPs are constructed free of feedback loops through the "*join*" function. Figure 15 shows how the *join* function works initially. As noted previously, feedback loops do not contribute to the critical path of CN and may be eliminated. This being the case, the resulting EPAs constructed from a finite collection of EPAs will be finite and the number of possible paths, P, spanning a component from input to output will be restricted by a finite set E. Some bounds on the process of forming pin-to-pin critical paths may be realized.

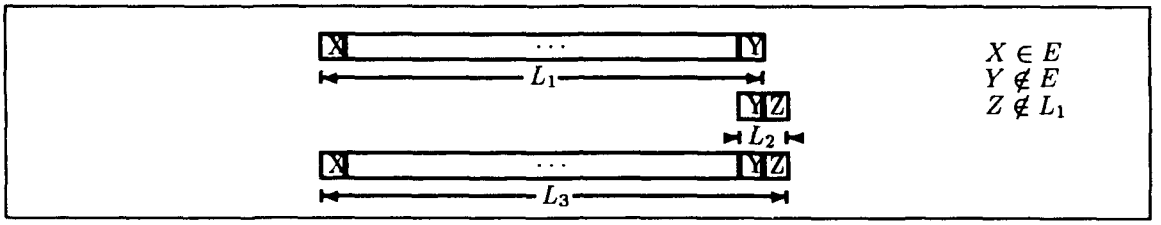


Figure 15. Initial Application of the *join* Function.

Figure 16 shows how the *join* function works at some point after the initial *join* function is used. Examining L_2 in Figure 16, it appears that cycles might exist within L_3 after the *join* operation is performed; however, the nodes forming the EAP between Y and Z are the interior nodes of the subcomponent to which the path L_2 belongs. Essentially, L_2 may be considered to have only Y and Z as shown in Figure 15. Thus, Theorem 1 and Lemma 1 still hold regardless of the level of hierarchy.

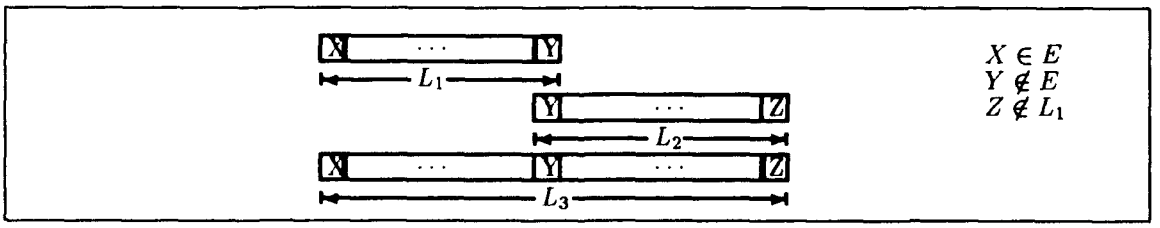


Figure 16. General Application of the *join* Function.

Within an extraction rule, there may exist more than one path for each subcomponent. Before any path manipulation is performed for a component, the subcomponents and interconnections are

checked by the extraction. Afterwards, a list of the component's external signals is constructed. All of the paths from the subcomponents are appended together into a list of paths. The list of paths and external signals is then passed to a Prolog function.

At the top level, the algorithm for finding the pin-to-pin critical paths is as follows.

initial conditions: a list, E, of external nodes and a list of paths.

1. Generate all possible new paths.
2. Eliminate any path for which the head or tail of the list is not in E.
3. Eliminate paths that are not pin-to-pin critical paths.

The algorithm for generating new paths is the following.

initial conditions: a list, E, of external nodes and a list of paths.

1. Perform the join operation on all lists of paths and add their respective delays.
2. Eliminate paths that were used to construct new paths where the heads of their EAPs are in E.
3. Repeat until no new paths are generated.

Once the extraction process has completed, another routine may be used to generate a structural VHDL description. The VHDL description is currently generated with pin-to-pin critical path information as comments.

Efficiency

Assume a component with I inputs, O outputs, and M nodes. Initially, all EAPs within the component will contain two nodes. In the worst case, all nodes might be interconnected. An enumeration of a worst-case initial condition is shown in Table 1. The "*" indicates no entry allowed[6]. From Table 1 the only lists that do not exist are those along the diagonal. This being the case, the worst-case initial number of EAPs is $m^2 - m$ or $O(m^2)$.

[6]Definition 2 in the previous section

Now we consider the worst case number of EAPs that might exist between two nodes. For the time being, consider only the upper-right half of Table 1 and an EAP with n_1 as an input and n_m as an output. From the table and the *join* function, the possible EAPs may be enumerated in the following manner.

$$
\begin{aligned}
L_0 &= [n_1, X_1^0, \ldots, X_{m-3}^0, X_{m-2}^0, n_m] \\
L_1 &= [n_1, X_1^0, \ldots, X_{m-3}^0, X_{m-2}^1, n_m] \\
L_2 &= [n_1, X_1^0, \ldots, X_{m-3}^1, X_{m-2}^0, n_m] \\
L_3 &= [n_1, X_1^0, \ldots, X_{m-3}^1, X_{m-2}^1, n_m] \\
&\vdots \\
L_{2^{m-2}-1} &= [n_1, X_1^1, \ldots, X_{m-3}^1, X_{m-2}^1, n_m]
\end{aligned}
$$

In each EAP, X_j^i designates the existence, X_j^1, or nonexistence, X_j^0, of its respective node, n_j+1. In the case X_j^0, the corresponding comma is not considered to exist in the EAP. From the enumeration of possible EAPs that may exist from n_1 to n_m, the order is $O(2^{m-2})$.

Table 1. Initial EAPs in a Hypothetical Component

node	n_1	n_2	n_3	...	n_m
n_1	*	$[n_1, n_2]$	$[n_1, n_3]$	...	$[n_1, n_m]$
n_2	$[n_2, n_1]$	*	$[n_2, n_3]$	...	$[n_2, n_m]$
n_3	$[n_3, n_1]$	$[n_3, n_1]$	*	...	$[n_2, n_m]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n_m	$[n_m, n_1]$	$[n_m, n_2]$	$[n_m, n_3]$	...	*

If we include the bottom-left half of the matrix, the problem becomes more complex. Since the interior nodes may now occur in any order and appear no more than once, the complexity of the problem is raised to $O((2^{m-2})!)$. This upper bound is highly pessimistic. Normal circuits do

not contain this high level of interconnectivity. The number of EAPs is further constrained by the scope of the extraction rule.

Since the efficiency of the join function is $O((2^{m-2})!)$, the join function is highly unreasonable as a tool for pin-to-pin critical path analysis, even for a modest number of interconnected nodes. However, within the realm of the GES extraction rule, the size of the circuit being examined is usually small. As a result, the number of interconnected nodes is small. The hierarchical nature of VHDL allows a circuit to be viewed as a component constructed of subcomponents which are, in turn, constructed from subsubcomponents and so forth, until the lowest level is reached. The extraction system uses the hierarchical view of the circuit, working from the lowest level toward the highest-level component view. As a result of the extraction process, a critical path of a component may be thought of as the construction of smaller critical paths through several of its subcomponents. Figure 17 shows how a pin-to-pin critical path is constructed from the pin-to-pin critical paths of its subcomponents. Therefore, it is not necessary to carry along other noncritical paths within subcomponents, thereby pruning out many nodes that would not contribute to the pin-to-pin critical path of the component.

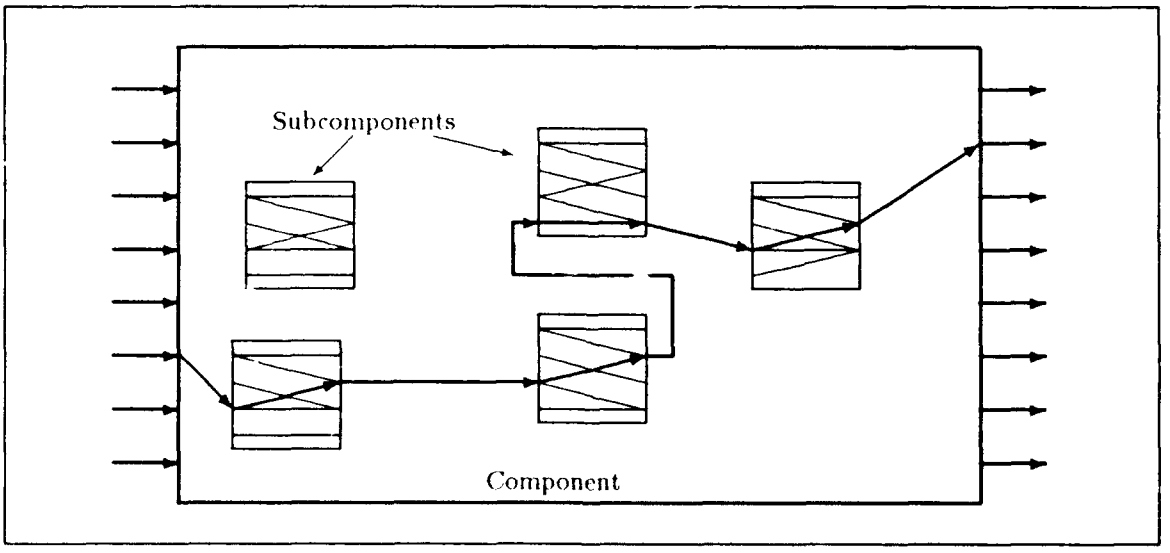


Figure 17. Critical Path Analysis.

False Paths

Though a discussion of all critical path analysis techniques is beyond the scope of this research, it is important to discuss one aspect of critical path analysis. The previous technique of calculating a pin-to-pin critical path for a component considers a full path regardless of whether the path through the subcomponents will contribute to the actual delay of the component. This type of path is referred to as a false path.

For a purely-combinational circuit, this type of problem may not arise; however, for a sequential circuit the result may be different. Should a pin-to-pin critical path be identified as in the previous section, there is no guarantee that there is a state in the hardware under consideration that would lead to the use of the identified path.

To illustrate a false path, consider a typical clocked JK flip-flop as shown in Figure 18. When the clock pulse on **c** is high, the inputs from **j**, **k**, **q**, and **notq** are able to propagate through the first stage, but not through the second stage. When the clock pulse on **c** is low, values are propagated through the second stage.

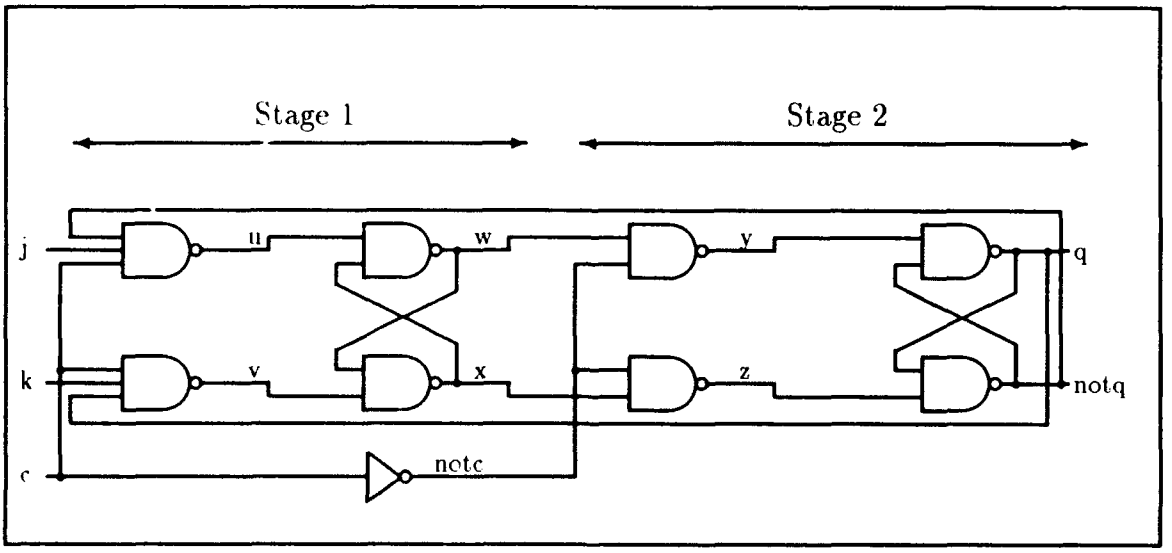


Figure 18. Typical Clocked JK Flip-Flop.

Assume that the propagation delay through the first two **NAND** gates is 4 nanoseconds (ns), the delay through the rest of the **NAND** gates is 3 ns, and the delay through the inverter is 1 ns. Knowing the delays through the components of Figure 18, a set of paths may be constructed. The paths are shown below.

```
[ [[notq,u],4], [[j,u],4], [[c,u],4], [[c,v],4], [[k,v],4], [[q,v],4],
  [[u,w],3], [[x,w],3], [[w,x],3], [[v,x],3], [[c,notc],1],
  [[w,y],3], [[notc,y],3], [[notc,z],3], [[x,z],3],
  [[y,q],3], [[notq,q],3], [[q,notq],3], [[z,notq],3] ]
```

The set of input and output nodes for the component is $E = j, k, c, q, notq$ which we will represent as the following.

```
[j,k,c,q,notq]
```

After application of the algorithm for finding the pin-to-pin critical paths, the following set of paths results for the clocked JK flip-flop.

```
[ [[notq,u,w,y,q],13], [[j,u,w,y,q],13], [[k,v,x,z,notq],13],
  [[q,v,x,z,notq],13], [[j,u,w,x,z,notq],16], [[c,u,w,x,z,notq],16],
  [[c,v,x,w,y,q],16], [[k,v,x,w,y,q],16] ]
```

If stage 1 is broken from stage 2 in Figure 18, a maximum delay path through stage 1 is `[[j,u,w],7]`, and a maximum delay path through stage 2 is `[[w,y,q],6]`. A maximum delay path through stage 1 will take 7 ns and a maximum delay path through stage 2 will take 6 ns. Putting both stages together renders a maximum delay for the clocked JK flip-flop of 13 ns; however, the pin-to-pin critical path analysis rendered a maximum delay path of `[[j,u,w,x,z,notq],16]` for a maximum delay of 16 ns through the clocked JK flip-flop. The maximum delay path of `[[j,u,w,x,z,notq],16]` is never used as a result of the clock.

Therefore, the rendered false path would provide an overly pessimistic view of the delay for a given pin-to-pin critical path. So the model presented in the previous section should be used with this caveat. If a more complicated model is desired, the previous section should be a guide on how to incorporate other critical path analysis models into the logic extraction process.

VII. Examples and Results

In this section, four layout designs in *magic* will be examined. The first is a rather simple design of a fabricated two-phase clock generator. The second design is an ALU that was verified using GES and fabricated. The third design is a 60,000 transistor design. The last design is a larger 250,000 transistor design. In all four examples, GES was used in the layout process to identify external and internal design errors. The first example demonstrates the Design-Rule Check (DRC) capability of GES. In the second example, a design is verified using only GES for functionality and critical path analysis. The third and fourth examples are used to demonstrate the performance of GES on large custom VLSI chip designs.

Clock Generator

A clock generator is an example of a circuit that functions correctly, yet cannot be simulated by *esim* (Terma 80). *esim* is a switch-level simulator that accepts as input a transistor netlist from *magic*. The simulator state advances after values on all nodes have converged to a steady state. Since the clock generator's normal function is to oscillate, *esim* cannot simulate the circuit; however, GES can extract the logical composition of the circuit to demonstrate that the correct components and the correct connections exist.

The transistors in the transistor netlist, generated from the mask layout description using *extract*, form fully static CMOS components. The logical components were extracted from the transistor netlist. The following is a listing of the log file.

```
| ?- ges.
finished with read.
Capacitor ptrans(nIZ_CAP2,n3_140_8,n3_140_8,136,52):removed
Capacitor ntrans(nIZ_CAP1,n3_12_115,n3_12_115,9,-42):removed
finished with find_error.
finished inverters.
finished tgates.
finished clk_inv.
finished nand.
finished nor.
finished find_more_errors.
```

There were two capacitors placed in the circuit to vary clock frequency and duty cycle. Both did not appear in the transistor netlist using *mextra* (Terma 86); however, they did appear using *extract* (Calif 86). The input to GES used the output from *extract*; thus the capacitively isolated signals, IZ_CAP1 and IZ_CAP2, show up in the report as capacitive transistors. The capacitively isolated signals **nIZ_CAP1** and **nIZ_CAP2** were found and removed. All other components in the circuit were successfully extracted using GES.

GES produced a listing of the components in the clock generator and their interconnections. The mask layout description of the clock generator was verified to have been generated correctly inasmuch as the components and their interconnections were concerned.

To introduce a flaw into the clock generator circuit, the metal-1 line for GND and the metal-1 line output of an inverter were shorted together, demonstrating a possible human error during layout. Figure 19 is a circuit diagram of the normal and abnormal portion of the affected circuit. To help make the example more interesting, a polysilicon line has also been severed. The first fault demonstrates a stuck-at-0 fault, whereas the second demonstrates a floating fault on one portion of the inverter. In addition to the external errors, an internal error is introduced between a **NOR** gate and an inverter. At this point there is a multiple fault in the circuit.

The following is a log of the Prolog session using GES on the errant clock generator circuit.

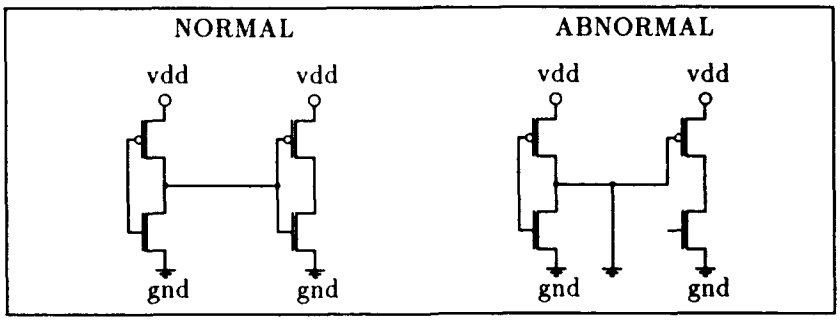


Figure 19. Circuit Diagram of Normal and Abnormal Circuit.

```
| ?- ges.
finished with read.
Bad trans ptrans(n3_12_115,nvdd,ngnd,70,7):removed
Straight wire ptrans(ngnd,nvdd,n3_44_29,99,8):removed
Capacitor ptrans(nIZ_CAP2,ngnd,ngnd,136,52):removed
Capacitor ntrans(nIZ_CAP1,n3_12_115,n3_12_115,9,-42):removed
Capacitor ntrans(n3_12_115,ngnd,ngnd,71,-21):removed
finished with find_error.
finished inverters.
finished tgates.
finished clk_inv.
finished nand.
Failure extracting component nor_gate(n3_72_106,
  n3_340_35,n3_238_103,155,11,1).  Internal node,
  n3_314_22, connected to an inverter at X:197, Y:13.
finished nor.
finished find_more_errors.
```

ALU

The design and construction of the ALU started with the VHDL design and ended with verification of the layout description. The steps in the design of the ALU are shown below.

1. A Behavioral VHDL description was generated.
2. A Structural VHDL description was generated.
3. Both the structural and behavioral descriptions were simulated together and results compared.
4. Corrections were made to the structural description as deviations were encountered.

5. *vhdl2ges* was run to generate a "customized" GES extraction system from the structural VHDL description for the design.
6. The physical layout description was generated by hand from the transistor level up.
7. As each cell was created, the "customized" GES extraction system was run to ensure conformance to the structural VHDL description.
8. Once the entire ALU was completed and verified using the "customized" GES extraction system, pin-to-pin critical path analysis was performed.

At no time in the design was a switch-level simulation or SPICE simulation of the layout design performed. During the layout process, various errors in interconnections were caught by the "customized" GES extraction system.

The ALU had a 4-bit opcode, 4-bit operands, 4/8-bit result, and comparator output. A view of the layout for the ALU is shown in Figure 20. The design was primarily based on the bit-slice design from (Mano 82:217-228) with the addition of a multiplier. Less than 2 man-weeks were required to fully describe the design in VHDL and lay out in *magic*. A total of 4 integrated-circuit chips were fabricated from MOSIS. All chips were tested with 80,000 test vectors in various sorted and psuedo-random sequences and were found to be 100% functionally correct. The pin-to-pin critical path analysis reported a 10MHz operation speed compared to the 13.5MHz actual speed of the chips.

60,000 Transistor Design

GES was used on a 60,000-transistor design of a multiplier section in a floating-point multiplier chip. The `statistic/0` function of Quintus Prolog was used to report periodic timing statistics within the extraction process. These results appear in Table 2. Less than 8 Megabytes of memory were required for this design. The performance results were collected from a MicroVAX 3600 running Ultrix V3.1 using Quintus Prolog. Three-hundred-eighteen design errors were found in the

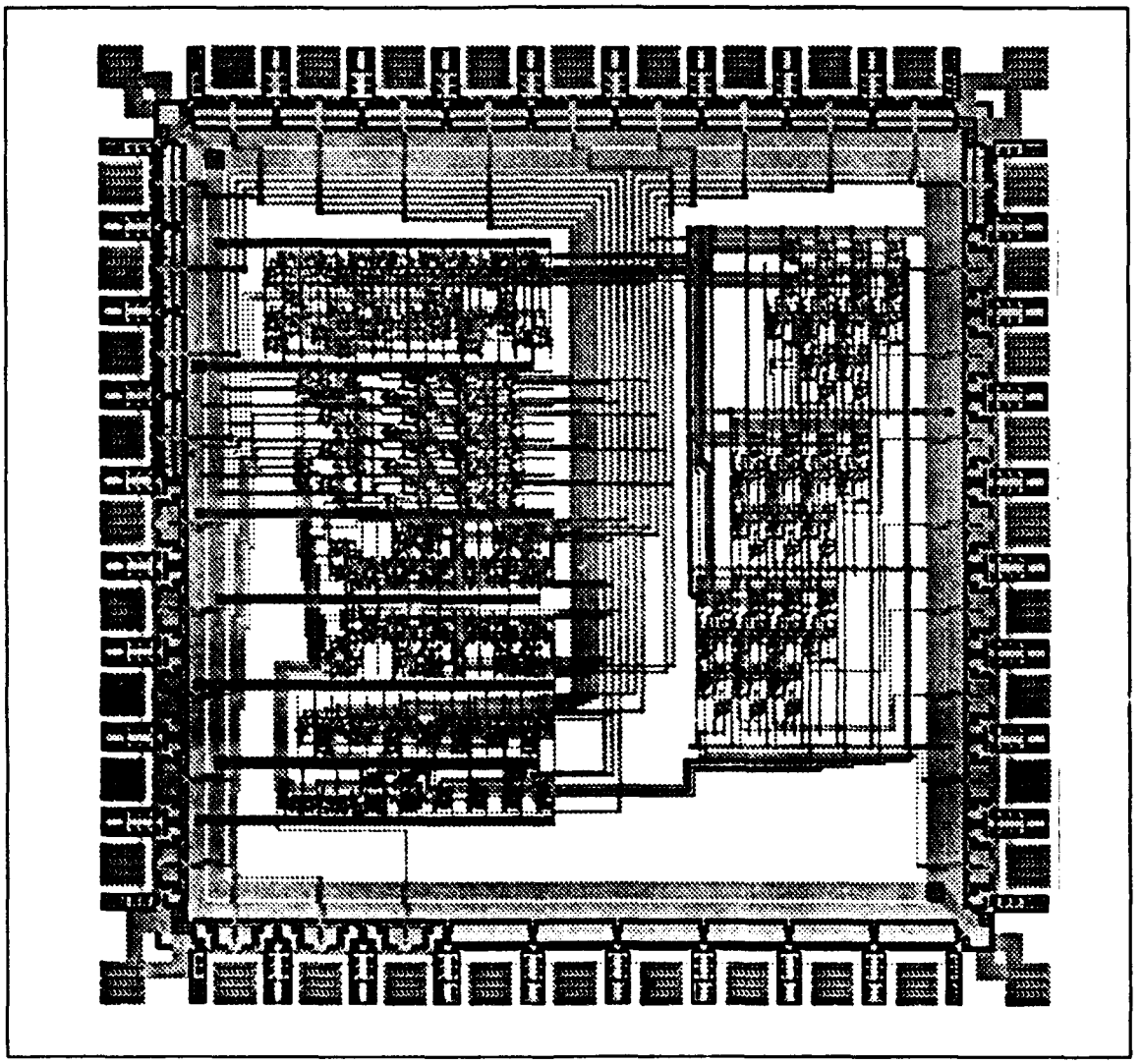


Figure 20. Layout of the ALU Integrated Circuit.

VLSI design. From Definition E1 (shown in Figure 9), there were 30 type-1 errors, 36 type-3 errors, 10 type-4 errors, 10 type-5 errors, and 120 type-6 errors. From Definition E2 (shown in Figure 10), there were 82 type-4 errors.

Time to Perform	CPU Time (min:sec)
Read and Eliminate Duplicates	025:40
Find Definition E1 Errors	000:26
Find inv	002:56
Find tgates	683:15
Find invZ	121:32
Find nand	036:30
Find nor	025:21
Find Definition E2 Errors	000:05
Find dff	000:11
Write Component Netlist	007:24

Table 2. Performance on a 60,000-Transistor Design

From the performance measurement of GES in Table 2 we can make a few observations. I/O for the example had moderate impact on the execution time of GES. Looking for errors in the design can be done quickly. This is indeed desirable, since a designer might want to interact with GES in an attempt to find errors that may exist within a layout design. Though the timing statistics do not represent many level-N components, extraction for other types of components (e.g., half-adders, adders, adder-arrays, registers) usually progresses quickly. The vast majority of extraction time is usually spent with the level-1 rules.

Performance on a 250,000-Transistor Design

A 250,000 transistor-sized section of a 1,500,000 transistor-sized design was used to examine the performance of the logic extraction rules when considering indexing. The inverter, transmission-gate, and clocked-inverter rules were used. The section used was largely constructed of memory cells. The platform used to perform extraction was a MicroVAX 3900 running Quintus Prolog

V2.4 on Ultrix V4.2. Shown in Table 3 are the results of running logic extraction on this 250,000 transistor-sized design.

Time to Perform	CPU Time (min:sec)	Memory (MegaBytes)
Read and Eliminate Duplicates	270:25	24
Find Definition E1 Errors	002:43	24
Find inv	012:54	24
Find tgates	028:42	24
Find invZ	018:12	24
Write Component Netlist	032:42	24

Table 3. Execution Times of Indexed Logic Extraction Rules for a 250,000 Transistor Design

The CPU time for extracting inverters, transmission-gates, and clocked inverters was less than an hour. From Table 3, the time to read in the transistors, eliminate duplicates, and build the transistor-facts data base took considerable CPU time. Altogether, the process consumed 6:05:38 (h:mm:ss) of CPU time. The total CPU time required for a 60,000 transistor-sized design for the same operation was 14:01:13. Indexing allowed for extraction of a design of more than four times in size in less than half the time.

VIII. Limitations

Presently, there are two types of limitations that exist in using GES. The first type, alluded to earlier, involves acceptable VHDL code. The second type involves designs that are not easily extracted. After the discussion of the limitations, some suggestions for overcoming these limitations will be discussed.

Currently, only structural VHDL descriptions with signals of mode **in** and **out** are accepted. The modes **buffer** and **inout** cannot be handled while still ensuring correct critical path analysis as described previously. There do not appear to be problems with signals of mode **linkage**.

The second limitation specifically involves the types of component configurations that are recognized and extracted. Assume that an extraction rule exists for identifying an **AND** gate formed from a **NAND** gate followed by an inverter. A typical GES extraction rule for identifying this configuration is the following.

```
and_gate :-
  nand_gate(Nx,Ny,Ninterm,X0,Y0,_),
  inv(Ninterm,Noutput,X1,Y1,_),
  unique_component([[X1,Y1],[X0,Y0]]),
  not_connected([Ninterm],[Nx,Ny,Noutput]),
  retract(inv(Ninterm,Noutput,_,_,_)),
  retract(nand_gate(Nx,Ny,Ninterm,_,_,_)),
  find_anomaly_list(and_gate(Nx,Ny,Noutput,X0,Y0,1),[Ninterm]),
  assert(and_gate(Nx,Ny,Noutput,X0,Y0,1)),
  fail.
and_gate.
```

From the extraction rule **and_gate/0**, every **NAND** gate followed by an inverter will be replaced with an **AND** gate.

Consider an additional extraction rule, **half_adder_cc/0**, constructed as follows.

```
half_adder_cc :-
  xor_gate(Nx,Ny,Nsum,X0,Y0,_),
  nand_gate(Nx,Ny,Ncbar,X1,Y1,_),
  inv(Ncbar,Ncarry,X2,Y2,_),
  unique_component([[X2,Y2],[X1,Y1],[X0,Y0]]),
  not_connected([Ncbar],[Nx,Ny,Nsum,Ncarry]),
  retract(inv(Ncbar,Ncarry,_,_,_)),
  retract(nand_gate(Nx,Ny,Ncbar,_,_,_)),
  retract(xor_gate(Nx,Ny,Nsum,_,_,_)),
  find_anomaly_list(half_adder_cc(Nx,Ny,Nsum,Ncarry,X0,Y0,1),[Ncbar]),
  assert(half_adder_cc(Nx,Ny,Nsum,Ncarry,X0,Y0,1)),
  fail.
half_adder_cc.
```

We will use the **and_gate/0** and **half_adder_cc/0** rules to perform extraction on the following components.

```
nand_gate(na,nb,ncbar,1,1,1).
xor_gate(na,nb,nsum,10,1,1).
inv(ncbar,ncarry,1,10,1).
```

If the **half_adder_cc/0** extraction rule is used before the **and_gate/0** extraction rule, the following will result.

```
half_adder_cc(na,nb,nsum,ncarry,10,1,1).
```

However, if the **and_gate/0** extraction rule is used before the **half_adder_cc/0** extraction rule, the following will result.

```
xor_gate(na,nb,nsum,10,1,1).
and_gate(na,nb,ncarry,1,1,1).
```

There are three methods for solving this problem. The first method involves ordering the rules such that the **half_adder_cc/0** extraction rule is called before the **and_gate/0** extraction rule. This prevents the **and_gate/0** extraction rule from interfering with proper extraction of the **half_adder_cc/0** extraction rule. The next two methods are interrelated.

If higher-level structural VHDL descriptions are not making use of the fact that an and_gate VHDL description exists, then the and_gate VHDL description should be eliminated. However, if it is necessary to have an and_gate VHDL description, ensure that all higher-level structural VHDL descriptions take advantage of the and_gate VHDL description. This type of reasoning may require the designer to make more prudent use of VHDL-based models. However, the designer may employ another method of enriching the extraction rule set by simply adding an additional VHDL description for half adders using **AND** and **Exclusive-OR** gates.

The previously described problem is not the only case where hierarchical extraction may require some manual intervention. Some extraction rules might force extractions over component boundaries. An example of how this might occur follows.

The previously defined extraction rules, **and_gate/0** and **half_adder_cc/0**, will be used. Assume a new extraction rule for **half_adder/0**.

```
half_adder :-
  half_adder_cc(Nx,Ny,Nsum,Ncbar,X0,Y0,_),
  inv(Ncbar,Ncout,X1,Y1,_),
  unique_component([[X1,Y1],[X0,Y0]]),
  not_connected([Ncbar],[Nx,Ny,Nsum,Ncout]),
  retract(inv(Ncbar,Ncout,_,_,_)),
  retract(half_adder_cc(Nx,Ny,Nsum,Ncbar,_,_,_)),
  find_anomaly_list(half_adder(Nx,Ny,Nsum,Ncout,X0,Y0,1),[Ncbar]),
  assert(half_adder(Nx,Ny,Nsum,Ncout,X0,Y0,1)),
  fail.
half_adder.
```

The component list from earlier in the section will be used. For clarity, the same netlist is shown below.

```
nand_gate(na,nb,ncbar,1,1,1).
xor_gate(na,nb,nsum,10,1,1).
inv(ncbar,ncarry,1,10,1)
```

If the extraction is performed in the order, **and_gate/0**, **half_adder_cc/0**, and **half_adder/0**, the result will be the following.

```
xor_gate(na,nb,nsum,10,1,1).
and_gate(na,nb,ncarry,1,1,1).
```

The three methods presented carlier are also useful in solving this problem.

There is one type of extraction problem that requires greater consideration. Assume we define a four-input **AND** gate as shown in Figure 21(a). Using the new extraction rule for a four-input **AND** gate, we will extract the circuit shown in Figure 21(b). Once the extraction process has completed, two different interpretations may result. In one case, the extraction process might yield two four-input **AND** gates and one two-input **AND** gate. In the second case, the extraction process might yield one four-input **AND** gate and four two-input **AND** gates. Currently, the method for solving this problem is to exclude VHDL descriptions for models that contain homogeneous structure.

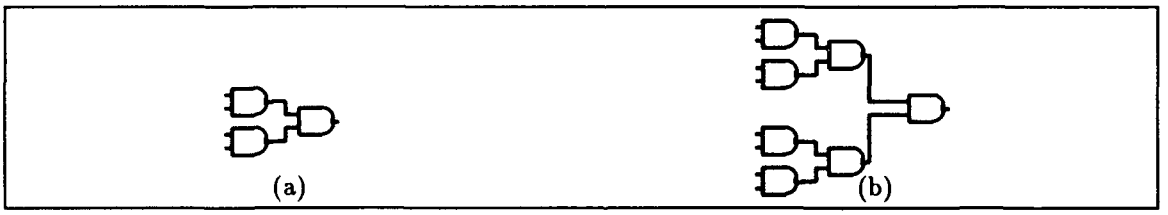


Figure 21. Four-Input **AND** Gate and Simple Circuit.

The limitations discussed in this chapter effect only the completeness of logic extraction. Because of these limitations, it is possible to have a structural specification that is equivalent to its layout specification, but not have the relation shown true by logic extraction. These limitations, however, do not effect the correctness of logic extraction.

IX. Conclusions and Recommendations

Conclusions

The objective of this research is to establish a formal definition of logic extraction, discuss properties of logic extraction as it relates to formal hardware-verification, and demonstrate that logic extraction is practical for VHSIC-class designs. We have established a definition of logic extraction in Prolog. In this manner, the syntax and semantics of logic extraction are established in a formal-executable frame-work. Further, properties of VHDL were identified and defined through a formal-executable frame-work. As such, properties of soundness and guaranteed termination were proven. Finally, pragmatic considerations for efficiency are met by taking advantage of the indexing trait of Quintus Prolog.

The practical aspects of logic extraction were demonstrated through the production of a working integrated circuit. Further, a methodology for employing logic extraction in the process of formally verifying the equivalence relation between a structural VHDL description and a layout description was exhibited. In addition to guaranteeing the equivalence between a structural VHDL description and a layout description, the required time to lay out a custom VLSI chip was reduced due to the diagnostic side-effect available through logic extraction.

This work has demonstrated that logic extraction is not as simple as previously thought. Logic extraction embodies implicit as well as explicit interconnection properties. Logic extraction may also limit the design problem space so that other diagnostic tools may be used to examine portions of a circuit to generate further information for the designer. Extracting pin-to-pin critical paths is but just one example.

Logic extraction is not just limited to custom VLSI. Logic extraction may be used to assist verification of the correctness of synthesis. One attribute of logic extraction is the ability to extract a circuit without the benefit of a structural VHDL description. This attribute may further aid in the future respecification of nonreprocurable digital parts.

Recommendations for Future Work

The research explored in this dissertation provides a basis for several new areas of research. Within this section, new areas of possible research are recommended.

Deriving logic extraction rules from an increased syntax of VHDL would be highly desirable. Structural VHDL was shown to supply sufficient basis for constructing logic extraction rules; however, the properties being proven by logic extraction also exist implicitly in data-flow VHDL. A tool for translating structural VHDL to data-flow VHDL is presented as an appendix and could be reviewed as a formal method for bridging the gap between structural VHDL and data-flow VHDL for logic extraction.

Now that critical path analysis has been demonstrated to be feasible within the framework of logic extraction, other forms of analysis similar to critical path analysis appear plausible. There appears to be potential for cell-by-cell power analysis of VLSI circuits. Computations based on worst-case analysis or a typical-case model of power consumption seem feasible. It is possible that some statistical methods may be used to predict power requirements closer to actual power requirements through extraction rather than a worst-case power requirements analysis.

Other forms of path analysis may be considered. Reliability analysis might be implemented through extraction, similar to critical-path analysis. Searching for the most unreliable path through a system appears feasible using a deviation of the Huffman model for critical-path analysis.

An extension to the logic extraction system might include a function extraction routine for domino (Weste 85:168) transistor networks. Logic extraction rules that are difficult to tailor to some transistor networks could be enhanced through a new type of logic extraction rule.

GES is scheduled for use on a million-transistor project. Portions of a multi-chip module will probably be synthesized. GES will be used to extract manually-generated portions of the design, i.e., the BIST portions (Seraf 91a) (Seraf 91b) of the design. In this manner, the manually-generated portions of the design can be verified and the synthesized portions extracted to a gate-level view

for simulation in VHDL. This project should provide a basis for studying the utility of combining logic extraction with synthesis in creating large integrated circuits. A methodology for such use may be generated from the experience gathered here.

Appendix A. Definitions

Definition of Behavioral and Structural Specification

In this section, a definition will be given for behavioral specification and structural definition. For the purposes of the presentation, combinational logic will be used. Afterwards, an example of each in VHDL will be offered to help distinguish the two from each other.

A behavioral specification is an algorithmic description of how a specified system or component is expected to react to a given set of input stimuli. There is usually nothing or very little provided in the behavioral specification as to the internal physical makeup and interconnections of the specified system of component. The behavioral specification may be represented abstractly as in Figure 22. The boundary of the specified system or component is well defined. By well defined we mean that all inputs and outputs are identified at the boundary of the specified device or component.

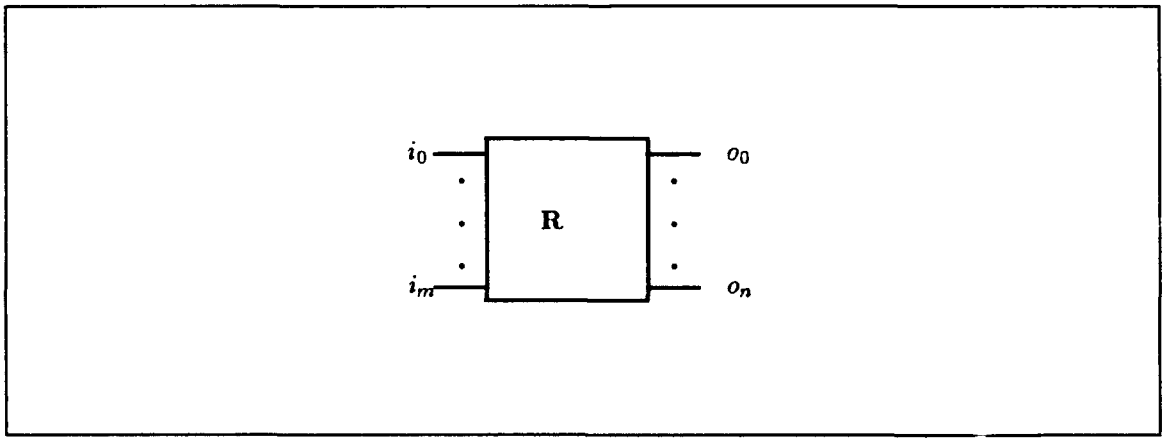


Figure 22. Abstract View of a Behavioral Specification.

From Figure 22, the inputs to the specified device or component are represented as I where $I = i_0, ..., i_m$ and $0 \leq m$. The outputs from the specified device or component are represented as O where $O = o_0, ..., o_n$ and $0 \leq n$. The algorithm for the specified device or component is represented by the relation $\mathbf{R}$ where $\mathbf{R} \subset I \times O$.

A structural specification for a specified device or component provides a description of the internal physical makeup and interconnections. The following criteria are used to determine a structural specification.

1. A structural specification is not a behavioral specification.
2. A structural specification may be constructed from one or more interconnected behavioral specifications.
3. A structural specification may be constructed from one or more interconnected structural specifications.
4. A structural specification may be constructed from one or more interconnected behavioral specifications and structural specifications.

The behavioral specification may be viewed as a procedural method for portraying a specified device or component. Alternatively, the structural specification may be viewed as a declarative method for portraying a specified device or component.

The following is a VHDL model of a behavioral specification.

```
entity adder is
  port(i0,i1,i2: in  bit;
       o0,o1   : out bit);
  end adder;

architecture behave of adder is

  function bv (input : bit) return integer is
    begin
    If(input = '1') then return(1);
    else return(0); end if;
    end;

  function bv_inv (input : integer) return bit is
    begin
    If(input = 0) then return('0');
    else return('1'); end if;
    end;

  begin

  process

    begin
    wait on i0,i1,i2;
    if ((bv(i0)+bv(i1)+bv(i2)) < 2) then
      o0 <= bv_inv(bv(i0)+bv(i1)+bv(i2));
    else
      o0 <= bv_inv(bv(i0)+bv(i1)+bv(i2)-2);
    end if;
    if ((bv(i0)+bv(i1)+bv(i2)) < 2) then
      o1 <= bv_inv(0);
    else
      o1 <= bv_inv(1);
      end if;
    end process;
  end behave;
```

From the VHDL description the inputs and output are enumerated as i0, i1, i2 for the inputs and o0,o1 for the outputs. The assigned values for the outputs are determined algorithmically from the inputs. There is nothing in the VHDL description that indicates the physical construction of the specified device or component.

The following VHDL description is a structural specification of the same device.

```
entity adder is
  port(i0,i1,i2: in  bit;
       o0,o1   : out bit);
  end adder;

architecture structure of adder is

  signal p,q,r : bit;

  component half_add
    port (a,b : in bit;
          s,cbar : out bit);
    end component;

  begin

  o1 <= q nand r;

  ha1 : half_add port map
    ( a => i0,
      b => i1,
      s => p,
      cbar => q);

  ha2 : half_add port map
    ( a => p,
      b => i2,
      s => o0,
      cbar => r);

  end structure;
```

Definitions for Other Terms

Hardware description language (HDL) "a language used to describe a circuit's behavior or structure." (deGeu 89:27)

Logic synthesis "creation of a gate-level netlist from a register-transfer level description." (deGeu 89:27)

Many-valued Logic an algebraic system consisting of more than two truth values (Resch 69:17).

Mapping "the process of formulating a design in terms of the cells available in a given parts library." (deGeu 89:27)

Netlist "a circuit design description in terms of structural elements and their interconnections."

(deGeu 89:27)

Appendix B. Efficiency Issues

Foremost in the design of GES was the concern for correct execution of extraction. For designs containing a large number of components, efficiency of the system does become a practical concern. In a sense, this appendix addresses the pragmatics of performing extraction.

Logic Extraction

The extraction process can occupy a large amount of CPU time. In order to help reduce the CPU time involved in extracting components, some heuristics are offered. The first heuristic identifies low-level signature components of higher level components. The second heuristic eliminates duplicate transistors where possible. The third heuristic seeks to reduce the complexity of extraction-rules. These three heuristics are explained below. Finally, the knowledge of term indexing in Quintus Prolog is used to increase the execution speed of logic extraction with minimal impact on memory.

Extraction Without Indexing

The complexity of logic extraction consists of two parts. These parts concern the number of components to be examined and the number of extraction rules used. For the purpose of discussion we will assume a component data base consisting of one single type of component. By a single type of component, we mean that the functor/arity of all components is the same. Assume that the component data base is of size n. Additionally, assume a logic extraction rule matching m components of the same functor/arity of the component data base. Assume also the process of logic extraction as described in Prolog for nonindexed terms in the component data base. Under such conditions, we can show the complexity is of $O(n^m)$.

Through induction, the following assertion will be proven

$$((E\ 1) \wedge (\forall m.(E\ (m-1)) \Rightarrow (E\ m))) \Rightarrow \forall m.(E\ m)$$

where E represents the case of an extraction rule matching m components. Considering the base case, an extraction rule must contain at least one matching component. The extraction rule will execute n times for n components in the data base since the backtracking mechanism of Prolog requires it. Therefore, for $(m = 1)$ we have $O(n)$.

Assume now the complexity of $O(n^{(m-1)})$ for the case of matching $(m-1)$ components. Under this assumption, it is necessary to examine the cause for an extraction rule to exhibit this behavior. We will use the following template for an extraction rule.

$$\begin{array}{l}
E :- \\
\quad C_1(T_1^1, T_2^1, \ldots, T_o^1), \\
\quad C_2(T_1^2, T_2^2, \ldots, T_o^2), \\
\quad\quad \vdots \\
\quad C_{(m-1)}(T_1^{(m-1)}, T_2^{(m-1)}, \ldots, T_o^{(m-1)}), \\
\quad \mathit{not_connected}(\mathit{Signal}_a, \mathit{Signal}_e), \\
\quad \mathit{retract}(C_1(T_1^1, T_2^1, \ldots, T_o^1)), \\
\quad \mathit{retract}(C_2(T_1^2, T_2^2, \ldots, T_o^2)), \\
\quad\quad \vdots \\
\quad \mathit{retract}(C_{(m-1)}(T_1^{(m-1)}, T_2^{(m-1)}, \ldots, T_o^{(m-1)})), \\
\quad \mathit{asserta}(E(\mathit{Signal}_e)), \\
\quad \mathit{fail}. \\
E.
\end{array}$$

In order to force backtracking complexity of $O(n^{(m-1)})$, each C_i, $1 \leq i \leq (m-1)$, would have to succeed followed by the failure of **not_connected/2**.

Finally, the m^{th} matching rule is added after $C_{(m-1)}(T_1^{(m-1)}, T_2^{(m-1)}, \ldots, T_o^{(m-1)})$. By the backtracking nature of Prolog, for every possible combination of the first $(m-1)$ matching rules, the m^{th} matching rule will be tried n times. By assumption, the first $(m-1)$ matching rules are executed $(n^{(m-1)})$ times. By multiplication we have $(n^{(m-1)}) \times n = n^m$ rendering a complexity of $O(n^m)$.

Signature Components

Clauses within Prolog rules are executed sequentially. Thus, the order in which clauses appear in Prolog rules can affect execution efficiency[1]. This procedural aspect of Prolog will be investigated below to help understand how it can be used to speed up extraction. Assume we are interested in increasing the execution efficiency of a particular Prolog rule, called $comp_1$ shown below.

$comp_1$:-
  $subcompA(A_1, A_2, \ldots, A_o)$,
  $subcompB(B_1, B_2, \ldots, B_p)$,
  $subcompC(C_1, C_2, \ldots, C_q)$,
  $retract(subcompA(A_1, A_2, \ldots, A_o))$,
  $retract(subcompB(B_1, B_2, \ldots, B_p))$,
  $retract(subcompC(C_1, C_2, \ldots, C_q))$,
  $asserta(comp_1(Co_1, Co_2, \ldots, Co_r))$,
  fail.
$comp_1$.

Assume that there are j *subcompA* components, k *subcompB* components, and l *subcompC* components. Should execution of the $comp_1$ rule lead to m $comp_1$ components exhausting all j, k, and l subcomponents, respectively, then $j = k = l = m$. Assume, now, the existence of another component, $comp_2$, with the rule

$comp_2$:-
  $subcompB(B_1, B_2, \ldots, B_p)$,
  $subcompC(C_1, C_2, \ldots, C_q)$,
  $retract(subcompB(B_1, B_2, \ldots, B_p))$,
  $retract(subcompC(C_1, C_2, \ldots, C_q))$,
  $asserta(comp_2(Co_1, Co_2, \ldots, Co_s))$,
  fail.
$comp_2$.

Assume, too, that n $comp_2$ components exist and that after the application of both rules, $comp_1$ and $comp_2$, all j, k, and l subcomponents will be exhausted. From the above extraction process, then, $j = m$ and $k = l = m + n$. Since *subcompA* occurs only in $comp_1$, *subcompA* is called the signature of $comp_1$.

[1]For those more familiar with Prolog, the terms *shallow* and *deep* backtracking may come to mind. For those wishing to know more about these terms, a discussion is found in (Sterl 1986)

Furthermore, extraction using the Prolog rule $comp_1$ before the Prolog rule $comp_2$ is preferred. If we were to use the Prolog rule $comp_2$ first, then all k *subcompB* components would be searched for inclusion in $comp_2$. Whereas, using the Prolog rule $comp_1$ first would reduce the search space to $k - m$ *subcompB* components for $comp_2$. The search rationale is further predicated on the assumption that some of the parameters for a subcomponent aid in the selection of subsequent subcomponents in a Prolog rule. To help understand how parameters aid in the selection of subsequent subcomponents consider the following explanation.

Assume that some parameter of *subcompA*, called A_h where $1 \leq h \leq o$, is connected to some parameter of *subcompB*, called B_i where $1 \leq i \leq p$, in $comp_1$. In the execution of the Prolog rule $comp_1$, Prolog will attempt to find a component in the component database called *subcompA* before looking for *subcompB*. Once a component is found satisfying *subcompA*, the parameters of *subcompA* will be instantiated (or unified) to the values corresponding to the component in the component database. Since $B_i = A_h$ in *subcompB*, B_i is instantiated to the value of A_h and will therefore constrain Prolog to finding a component that satisfies *subcompB* and B_i. The additional constraint of B_i reduces the possible components to be considered in satisfying $comp_1$.

Consider the following example using the Prolog rule described earlier for a D flip-flop:

```
dff :-
  clk_inv(P2,P1,G,X,Xloc,Yloc,1),
  tgate(P1,P2,D,X,_,_),
  inv(X,G,_,_,1),
  remove_tgate(P1,P2,D,X),
  retract(clk_inv(P2,P1,G,X,Xloc,Yloc,1)),
  retract(inv(X,G,_,_,1)),
  asserta(dff(P1,P2,D,G,Xloc,Yloc,1)),
  fail.
dff.
```

We may compare the **dff** rule to a new rule concerning **xor**.

```
xor :-
  tgate(B,Bnot,A,XOR,_,_),
  inv(B,Bnot,_,_,1),
  inv(A,Anot,_,_,1),
  tgate(Bnot,B,Anot,XOR,Xloc,Yloc),
  retract(inv(B,Bnot,_,_,1)),
  retract(inv(A,Anot,_,_,1)),
  remove_tgate(Bnot,B,Anot,XOR),
  remove_tgate(B,Bnot,A,XOR),
  asserta(xor(A,Anot,B,Bnot,XOR,Xloc,Yloc,3)),
  fail.
xor.
```

Both **dff** and **xor** share transmission gates and inverters; however, **clk_inv** only occurs in **dff**. In this case, **clk_inv** would be considered a signature component for **dff**. Also notice in **xor** that finding **tgate(B,Bnot,A,XOR,_,_)** would easily lead to location of one inverter, aid in the quick selection of a second, and to location of the second transmission gate.

Eliminating Duplicates

The second heuristic seeks to eliminate duplicate transistors. Since only digital logic is of interest, additional transistors (added to increase the drive capacity of a circuit) needlessly increase the search space. When only looking for the logic functionality of a circuit, no additional information is gained from such transistors. The following is a Prolog rule adopted to eliminate duplicate transistors while reading in the transistor netlist from a mask layout description.

```
remove_dup_trans :-
  read(X),
  remove_dup_trans(X),!,
  remove_dup_trans.
remove_dup_trans.
remove_dup_trans(end_of_file) :- !.
remove_dup_trans(p(A,B,C,_,_,_,_)) :-
  ptrans(A,B,C,_,_),!.
remove_dup_trans(n(A,B,C,_,_,_,_)) :-
  ntrans(A,B,C,_,_),!.
remove_dup_trans(p(A,B,C,W,L,X,Y)) :-
  asserta(p(A,B,C,W,L,X,Y)),!.
remove_dup_trans(n(A,B,C,W,L,X,Y)) :-
  asserta(n(A,B,C,W,L,X,Y)),!.
```

Reducing Prolog Rule Complexity

The third heuristic addresses rule complexity. Rule complexity is directly related to the number of components that must be matched. Therefore, rule complexity increases as the number of components that must be matched increases. In general, simpler rules increase execution efficiency.

An example of how rule complexity influences efficiency may be found in the identification of registers from a component netlist. Assume the following rule, *register1*, for registers.

```
register1 :-
  clk_inv(R,P,C1,C1bar,X,Y),
  inv(P,R,_,_),
  tgate(In,P,C1bar,C1,_,_),
  tgate(R,Q,C2bar,C2,_,_),
  clk_inv(S,Q,C2,C2bar,_,_),
  inv(Q,S,_,_),
  tgate(S,Out,Abar,A,_,_),
  retract(clk_inv(R,P,C1,C1bar,X,Y)),
  retract(inv(P,R,_,_)),
  retract(tgate(In,P,C1bar,C1,_,_)),
  retract(tgate(R,Q,C2bar,C2,_,_)),
  retract(clk_inv(S,Q,C2,C2bar,_,_)),
  retract(inv(Q,S,_,_)),
  retract(tgate(S,Out,Abar,A,_,_)),
  asserta(register(In,Out,C1,C1bar,C2,C2bar,A,Abar,X,Y)),
  fail.
register1.
```

Figure 23 is a diagram of the component extracted by the *register1* rule.

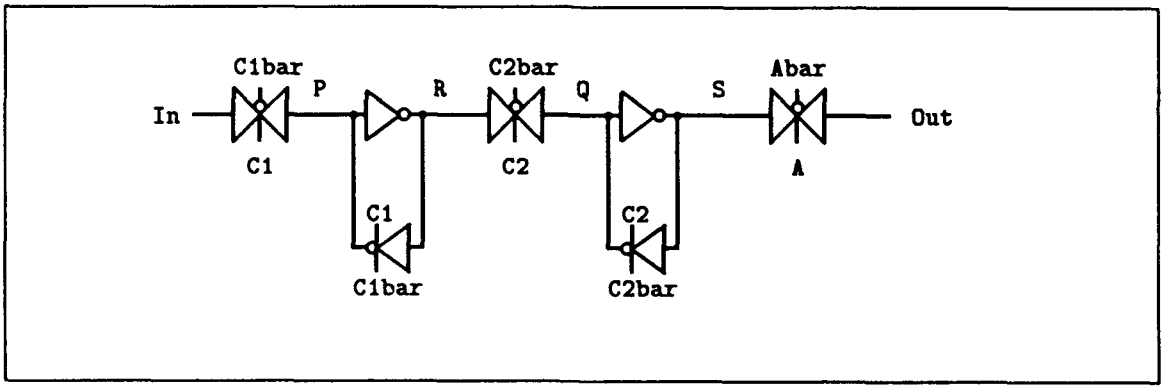


Figure 23. Schematic for *register1* Extraction Rule.

Assume a component netlist consisting of *clk_inv*, *inv*, and *tgate* that when extracted form j registers with no residual components. Assume also that a register is constructed from two D flip-flops, described below, and a transmission gate as in Figure 24 and by the Prolog rule for *register2* that follows.

```
register2 :-
  dff(In,R,C1,C1bar,X,Y),
  dff(R,S,C2,C2bar,_,_),
  tgate(S,Out,Abar,A,_,_),
  retract(dff(In,R,C1,C1bar,X,Y)),
  retract(dff(R,S,C2,C2bar,_,_)),
  retract(tgate(S,Out,Abar,A,_,_)),
  asserta(register(In,Out,C1,C1bar,C2,C2bar,A,Abar,X,Y)),
  fail.
register2.

dff :-
  clk_inv(R,P,C1,C1bar,X,Y),
  inv(P,R,_,_),
  tgate(In,P,C1bar,C1,_,_),
  retract(clk_inv(R,P,C1,C1bar,X,Y)),
  retract(inv(P,R,_,_)),
  retract(tgate(In,P,C1bar,C1,_,_)),
  asserta(dff(In,R,C1,C1bar,X,Y)),
  fail.
dff.
```

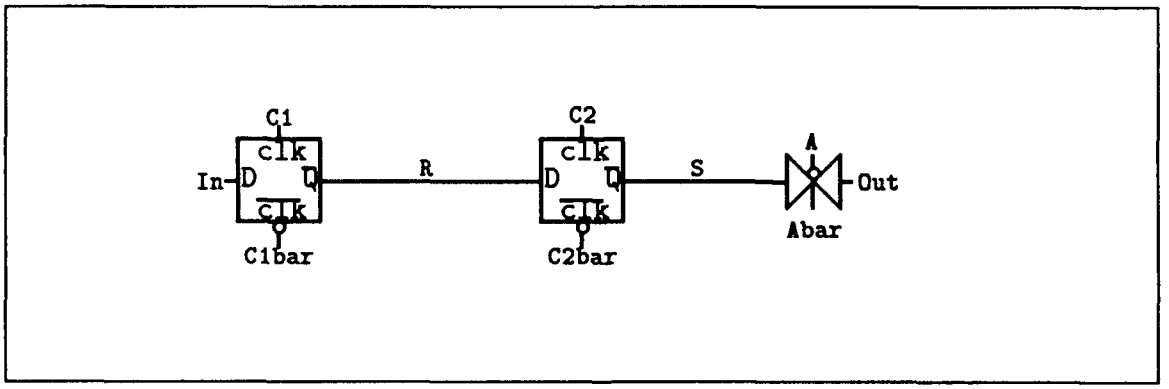


Figure 24. Schematic for *register2* Extraction Rule.

Using the rules *register2* and *dff* there are k *dff*, where $k = 2 * j$, and j *tgate*. If the rule *register1* is considered, there are k *clk_inv*, k *inv*, and l *tgate* where $l = j + k$.

For the purpose of the illustration consider *register1*, *dff*, and *register2* in the following manner. The parts of each rule that query the fact database will also be numbered to aid in the discussion.

```
            register1 :-
R11           clk_inv(R,P,C1,C1bar,X,Y),
R12           inv(P,R,_,_),
R13           tgate(In,P,C1bar,C1,_,_),
R14           tgate(R,Q,C2bar,C2,_,_),
R15           clk_inv(S,Q,C2,C2bar,_,_),
R16           inv(Q,S,_,_),
R17           tgate(S,Out,Abar,A,_,_),

            dff :-
D1            clk_inv(R,P,C1,C1bar,X,Y),
D2            inv(P,R,_,_),
D3            tgate(In,P,C1bar,C1,_,_),

            register2 :-
R21           dff(In,R,C1,C1bar,X,Y),
R22           dff(R,S,C2,C2bar,_,_),
R23           tgate(S,Out,Abar,A,_,_),
```

Statements R11, D1, and R21 may be considered as enumeration statements (or ENUMERATE) since they simply pick off from the database of facts the next available fact until all facts that satisfy predicate/arity have been exhausted. Statements R12...R17, D2, D3, R22, and R23, may be considered as database queries (or QUERY) since some or all of their parameters have been unified based upon the previous statements. If we also assume the worst-case ordering of components such that the first k *clk_inv* actually form the second *dff*, the rule *register1* will "fail" k times before it will actually begin identifying registers. Furthermore, the k times that *register1* failed it identified k *dff*. Using *register1* to identify registers, there will be at most k failed ENUMERATEs and $4 * k$ failed QUERYs. The failed QUERYs are incurred since R12, R13, and R14 succeed, but R15 will fail causing the entire sequence to backtrack and try a new *clk_inv*.

Consider the rules *dff* and *register2* on the same component netlist. The rule *dff* will succeed until all *clk_inv* have been exhausted. If we assume that the *dff* components were ordered in the worst case then *register2* will have only k failed ENUMERATEs and no more. The $4 * k$ failed QUERYs from *register1* were avoided by reducing its complexity.

Generally, the above three heuristics have been found to increase the speed of execution. Identifying signature components may be dependent on the composition of a given component netlist and should therefore be considered. Eliminating duplicate components not only reduces the search space but allows for parallelization of the extraction process. Finally, reducing rule complexity increases efficiency by reducing search failures.

Appendix C. Using HOL

Preliminaries

Theorems may be proven in HOL either interactively or through the ML "let" command. Interactive theorem proving is performed by entering line-by-line commands in order to manipulate an HOL goal stack and HOL assumption stack. Entering theorems in HOL through the ML "let" command inserts a proven theorem into a theory data file. Before HOL statements are presented, some symbol definitions will be provided from the HOL manual.

```
Infix operators (Gordo87:3)

"t1=t2"     is equivalent to  "= t1 t2"      (read as "t1 equals t2")
"t1,t2"     is equivalent to  ", t1 t2"      (read as "the pair (t1,t2)")
"t1/\t2"    is equivalent to  "/\ t1 t2"     (read as "t1 and t2")
"t1\/t2"    is equivalent to  "\/ t1 t2"     (read as "t1 or t2")
"t1==>t2"   is equivalent to  "==> t1 t2"    (read as "t1 implies t2")
"t1<=>t2"   is equivalent to  "<=> t1 t2"    (read as "t1 iff t2")

Binders (Gordo87:3)

"!x.t"      is equivalent to  "!(\x.t)"      (read as "for all x, t")
"?x.t"      is equivalent to  "?(\x.t)"      (read as "for some x, t")
"@x.t"      is equivalent to  "@(\x.t)"      (read as "an x such that t")
```

Notice that several symbols are represented through combinations of characters. The $\wedge$ is represented in HOL by /\ for logical conjunction. The $\vee$ is represented in HOL by \/ for logical disjunction. The $\Rightarrow$ is represented in HOL by ==> for logical implication. The $\Leftrightarrow$ is represented in HOL by <=> for equivalence. The $\forall$ is represented in HOL by ! for universal quantification. The $\exists$ is represented in HOL by ? for existential quantification. Finally, the λ is represented in HOL by \ for lambda notation. HOL prompts the user for input through the use of a # prompt. The ;; is used to terminate an HOL command.

Showing Structure Implies Behavior Through HOL

The device in Figure 25 is a three-input component with a single output. The figure illustrates several methods for specifying the behavior of the same device. The behavioral specification representations shown are VHDL, HOL, a Karnaugh Map, and a Truth Table. The translation of VHDL into HOL is considered part of this research.

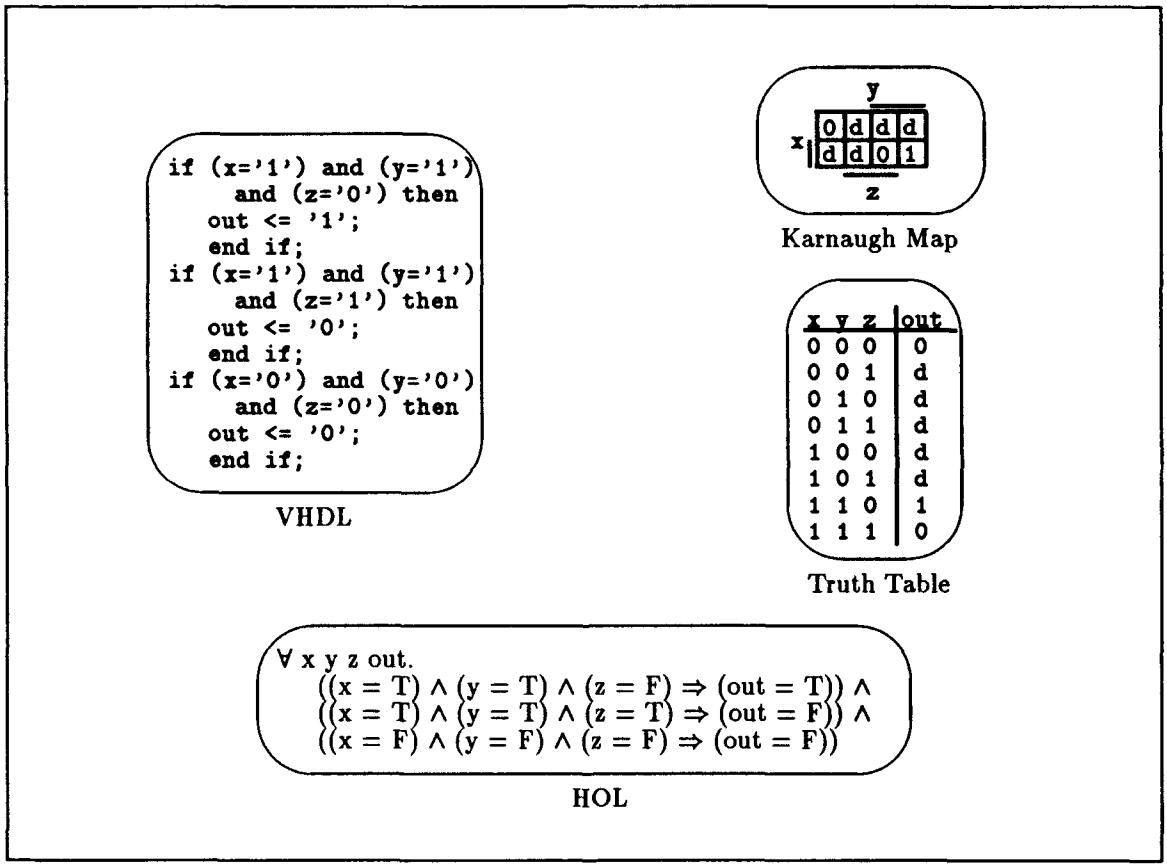


Figure 25. Behavioral Specifications for a Three-Input Device.

Three structural specification representations are illustrated in Figure 26. For each structural specification, the gate description, VHDL description, and HOL description are shown. However, nothing is known about the relation between the structural specifications shown in Figure 26 and the behavioral specification shown in Figure 25.

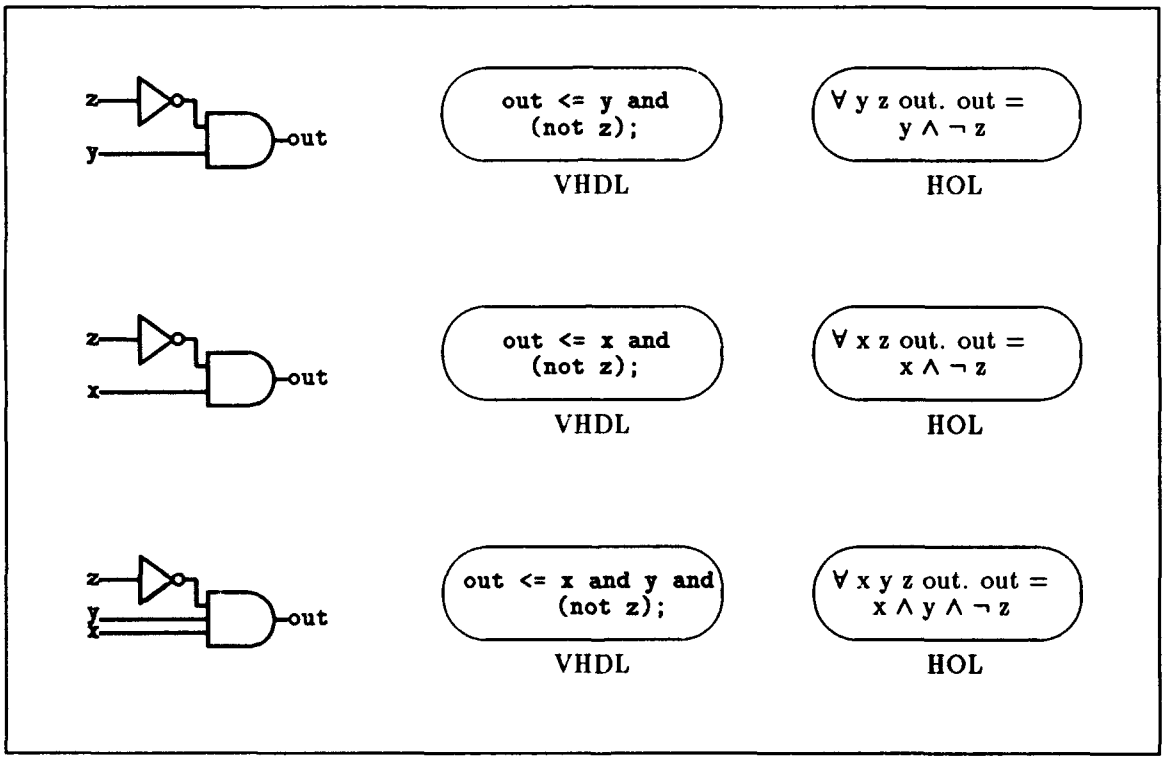


Figure 26. Three Implementation Specifications.

Before a comparison can be made between the behavioral specification and the structural specifications, a common specification language must be chosen. For the purpose of this research, the basis language will be VHDL. However, the VHDL descriptions must be transformed to another specification language to perform formal hardware-verification. Since HOL is to be used as the proof mechanism, the VHDL descriptions must be translated to HOL.

From Figures 25 and 26 the following HOL definitions were written for the behavioral specification and the three implementation specifications.

```
let behave_spec =
    new_definition('behave_spec',
        "!x y z out. behave_spec x y z out =
            (((x = T) /\ (y = T) /\ (z = F) ==> (out = T)) /\
             ((x = T) /\ (y = T) /\ (z = T) ==> (out = F)) /\
             ((x = F) /\ (y = F) /\ (z = F) ==> (out = F)))");;

let impl1_spec =
    new_definition('impl1_spec',
        "!x y z out. impl1_spec x y z out =
            (out = x /\ y /\ ~z)");;

let impl2_spec =
    new_definition('impl2_spec',
        "!x (y:bool) z out. impl2_spec x y z out =
            (out = x /\ ~z)");;

let impl3_spec =
    new_definition('impl3_spec',
        "!y z out. impl3_spec y z out =
            (out = y /\ ~z)");;
```

The HOL definition is the method for writing behavioral specifications and structural specifications. In order to establish that the structural specifications will perform as described by the behavioral specification, a proof will be constructed in HOL. For each structural specification, a theorem will be generated stating that for all inputs and outputs, the structural specification implies the behavioral specification. The first theorem,

$$\forall\ x\ y\ z\ out.\ impl1\_spec\ x\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out, \tag{13}$$

is shown in HOL below. Everything contained within quotes is considered to be the theorem.

```
#set_goal([],"!x y z out.
    impl1_spec x y z out ==> behave_spec x y z out");;
"!x y z out. impl1_spec x y z out ==> behave_spec x y z out"

() : void

#
```

The next step in the proof process is to remove the universal quantifier through a process in HOL called generalization. The procedure for manipulating the theorem to be proved in HOL is performed through a utility called **expand**. The expand utility provides a buffer between the user and the theorem to be proved. This prevents the user from performing an incorrect manipulation of the proof. In HOL, procedures called tactics are passed through **expand** to tell HOL how the theorem is to be modified. In order to generalize the universally quantified variables, the **GEN_TAC** tactic is used. Since **GEN_TAC** only generalizes one variable at a time, a modifier called **REPEAT** is used to perform **GEN_TAC** until it fails.

```
#expand(REPEAT GEN_TAC);;
OK..
"impl1_spec x y z out ==> behave_spec x y z out"

() : void

#
```

The next step is to replace **impl1_spec** and **behave_spec** with their definitions. This process is performed using the **REWRITE_TAC[]** tactic.

```
#expand(REWRITE_TAC[impl1_spec;behave_spec]);;
OK..
"(out = x /\ y /\ ~z) ==>
 (x /\ y /\ ~z ==> out) /\
 (x /\ y /\ z ==> ~out) /\
 (~x /\ ~y /\ ~z ==> ~out)"

() : void

#
```

At this point, the antecedent of the implication is assumed true through a tactic called **STRIP_TAC**. This process will create a list of assumptions or append to a list of assumptions should a list exist. Everything that appears between [ ] is considered to be an assumption.

```
#expand(STRIP_TAC);;
OK..
"(x /\ y /\ ~z ==> out) /\
 (x /\ y /\ z ==> ~out) /\
 (~x /\ ~y /\ ~z ==> ~out)"
    [ "out = x /\ y /\ ~z" ]

() : void

#
```

The **ASM_REWRITE_TAC[]** is different from the **REWRITE_TAC[]** in that substitutions will be performed within the theorem based upon the assumptions. The substitutions are performed by matching the left-hand side of the assumption with some element within the theorem and rewriting that element within the theorem with the right-hand side of the assumption.

```
#expand(ASM_REWRITE_TAC[]);;
OK..
"(x /\ y /\ z ==> ~(x /\ y /\ ~z)) /\
 (~x /\ ~y /\ ~z ==> ~(x /\ y /\ ~z))"
    [ "out = x /\ y /\ ~z" ]

() : void

#
```

The **ASM_REWRITE_TAC[]** and **REWRITE_TAC[]** tactics also perform other functions. One such function is to recognize tautologies through simple pattern matching. In this case, the **ASM_REWRITE_TAC[]** tactic eliminated **(x /\ y /\ ~z ==> x /\ y /\ ~z)** after the substitution.

The next step is to break up the theorem into two simpler theorems. The **CONJ_TAC** is used to create two separate theorems from a larger theorem formed by the conjunction of two separate theorems.

```
#expand(CONJ_TAC);;
OK..
2 subgoals
"~x /\ ~y /\ ~z ==> ~(x /\ y /\ ~z)"
    [ "out = x /\ y /\ ~z" ]

"x /\ y /\ z ==> ~(x /\ y /\ ~z)"
    [ "out = x /\ y /\ ~z" ]

() : void

#
```

The theorem that is now being manipulated is the bottom one in the previous list. Once again, the **STRIP_TAC** tactic is used to assume the antecedent of the implication true.

```
#expand(STRIP_TAC);;
OK..
"~(x /\ y /\ ~z)"
    [ "out = x /\ y /\ ~z" ]
    [ "x" ]
    [ "y" ]
    [ "z" ]

() : void

#
```

When an assumption is a literal, the literal is assumed true. Therefore, every occurrence of the literal in the theorem will be replaced with a "true" value when using the **ASM_REWRITE_TAC[]** tactic.

```
#expand(ASM_REWRITE_TAC[]);;
OK..
goal proved
... |- ~(x /\ y /\ ~z)
|- x /\ y /\ z ==> ~(x /\ y /\ ~z)

Previous subproof:
"~x /\ ~y /\ ~z ==> ~(x /\ y /\ ~z)"
    [ "out = x /\ y /\ ~z" ]

() : void

#
```

Once a theorem has been shown to be true, HOL responds with "goal proved" and a recapitulation of the theorems manipulated up to the point of proving it true. Should any other theorems remain to be shown, HOL will present them to the user. The "Previous subproof:" reply tells the user which theorem is to be proven next. As before, the **STRIP_TAC** is employed to assume the antecedent true.

```
#expand(STRIP_TAC);;
OK..
"~(x /\ y /\ ~z)"
    [ "out = x /\ y /\ ~z" ]
    [ "~x" ]
    [ "~y" ]
    [ "~z" ]

() : void

#
```

At this point, an **ASM_REWRITE_TAC[]** tactic will complete the proof.

```
#expand(ASM_REWRITE_TAC[]);;
OK..
goal proved
... |- ~(x /\ y /\ ~z)
|- ~x /\ ~y /\ ~z ==> ~(x /\ y /\ ~z)
|- (x /\ y /\ z ==> ~(x /\ y /\ ~z)) /\
   (~x /\ ~y /\ ~z ==> ~(x /\ y /\ ~z))
. |- (x /\ y /\ ~z ==> out) /\
     (x /\ y /\ z ==> ~out) /\
     (~x /\ ~y /\ ~z ==> ~out)
|- (out = x /\ y /\ ~z) ==>
   (x /\ y /\ ~z ==> out) /\
   (x /\ y /\ z ==> ~out) /\
   (~x /\ ~y /\ ~z ==> ~out)
|- impl1_spec x y z out ==> behave_spec x y z out
|- !x y z out. impl1_spec x y z out ==> behave_spec x y z out

Previous subproof:
goal proved
() : void

#
```

Once all of the theorems have been shown to be true, HOL responds by showing all of the theorems that were generated, to include the first theorem. At this point, it has been shown that

$$\forall\ x\ y\ z\ out.\ impl1\_spec\ x\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out. \tag{14}$$

Two more structural specifications remain to be examined. The next HOL proof will demonstrate how HOL tactics may be combined to shorten the proof process.

The second HOL proof will show

$$\forall\ x\ y\ z\ out.\ impl2\_spec\ x\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out. \tag{15}$$

```
#set_goal([],"!x y z out.
#    impl2_spec x y z out ==> behave_spec x y z out");;
"!x y z out. impl2_spec x y z out ==> behave_spec x y z out"

() : void

#
```

Experience with the first HOL proof would suggest that two HOL tactics could be used on the present theorem. The **REPEAT GEN_TAC** and **REWRITE_TAC[]** tactics may be invoked through the same **expand** HOL command. This is possible by the use of **THEN**. **THEN** works by applying the tactic on the left-hand side to the theorem first followed by the right-hand side tactic.

```
#expand(REPEAT GEN_TAC THEN REWRITE_TAC[impl2_spec;behave_spec]);;
OK..
"(out = x /\ ~z) ==>
 (x /\ y /\ ~z ==> out) /\
 (x /\ y /\ z ==> ~out) /\
 (~x /\ ~y /\ ~z ==> ~out)"

() : void

#
```

Continuing to Draw upon experience from the previous proof, the remainder of the HOL proof is performed below.

```
#expand(STRIP_TAC THEN ASM_REWRITE_TAC[]);;
OK..
"(x /\ y /\ ~z ==> x /\ ~z) /\
 (x /\ y /\ z ==> ~(x /\ ~z)) /\
 (~x /\ ~y /\ ~z ==> ~(x /\ ~z))"
    [ "out = x /\ ~z" ]

() : void

#expand(CONJ_TAC);;
OK..
2 subgoals
"(x /\ y /\ z ==> ~(x /\ ~z)) /\ (~x /\ ~y /\ ~z ==> ~(x /\ ~z))"
    [ "out = x /\ ~z" ]

"x /\ y /\ ~z ==> x /\ ~z"
    [ "out = x /\ ~z" ]

() : void

#expand(STRIP_TAC THEN ASM_REWRITE_TAC[]);;
OK..
goal proved
|- x /\ y /\ ~z ==> x /\ ~z

Previous subproof:
"(x /\ y /\ z ==> ~(x /\ ~z)) /\ (~x /\ ~y /\ ~z ==> ~(x /\ ~z))"
    [ "out = x /\ ~z" ]

() : void

#expand(CONJ_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC[]);;
OK..
goal proved
|- (x /\ y /\ z ==> ~(x /\ ~z)) /\ (~x /\ ~y /\ ~z ==> ~(x /\ ~z))
|- (x /\ y /\ ~z ==> x /\ ~z) /\
   (x /\ y /\ z ==> ~(x /\ ~z)) /\
   (~x /\ ~y /\ ~z ==> ~(x /\ ~z))
|- (out = x /\ ~z) ==>
   (x /\ y /\ ~z ==> out) /\
   (x /\ y /\ z ==> ~out) /\
   (~x /\ ~y /\ ~z ==> ~out)
|- !x y z out. impl2_spec x y z out ==> behave_spec x y z out

Previous subproof:
goal proved
() : void

#
```

The previous two HOL proofs demonstrate several points. The first and obvious point is that

$$\forall\ x\ y\ z\ out.\ impl1\_spec\ x\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out \tag{16}$$

and

$$\forall\ x\ y\ z\ out.\ impl2\_spec\ x\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out \tag{17}$$

demonstrate how a structural specification can be shown through a formal proof to satisfy a behavioral specification. Secondly, several proof steps may be combined into one step. The final point is that the choice of tactic is determined by the appearance of the theorem.

The two previous proofs were performed using tactics available in an older version of HOL. The most recent release of HOL was obtained within a week prior to this report. A new library has been added to HOL that includes a number of new tactics for propositional calculus. One new tactic, called **TAUT_TAC**, determines if a theorem is an instance of a tautology of the propositional calculus. The last proof of

$$\forall\ x\ y\ z\ out.\ impl2\_spec\ y\ z\ out \Rightarrow\ behave\_spec\ x\ y\ z\ out \tag{18}$$

will demonstrate its use.

```
#set_goal([],"!x y z out.
#    impl3_spec y z out ==> behave_spec x y z out");;
"!x y z out. impl3_spec y z out ==> behave_spec x y z out"

() : void

#load_library `taut`;;
Loading library `taut` ...
;; Fast loading file "/usr2/hol/Library/taut/taut_ml.o"
.....................
Library `taut` loaded.
() : void

#expand(REPEAT GEN_TAC THEN REWRITE_TAC[impl3_spec;behave_spec]);;
OK..
```

```
"(out = y /\ ~z) ==>
 (x /\ y /\ ~z ==> out) /\
 (x /\ y /\ z ==> ~out) /\
 (~x /\ ~y /\ ~z ==> ~out)"

() : void

#expand(TAUT_TAC);;
OK..
goal proved
|- (out = y /\ ~z) ==>
   (x /\ y /\ ~z ==> out) /\
   (x /\ y /\ z ==> ~out) /\
   (~x /\ ~y /\ ~z ==> ~out)
|- !x y z out. impl3_spec y z out ==> behave_spec x y z out

Previous subproof:
goal proved
() : void

#
```

Demonstrated within this section is one formal hardware-verification methodology. In this case, the behavioral and structural specifications were written in VHDL. In order to prove that the structural specifications implied the behavioral specification, it was necessary to translate the VHDL specifications into HOL definitions. The HOL definitions were then used to form theorems. The theorems were then proven through the use of tactics in HOL. The proof, in each case, formally verified that the structural specification implied the behavioral specification. This process illustrates how VHDL descriptions may be compared through HOL.

Appendix D. Translating Data Flow to Structure

Introduction

The purpose here is to present plausible mappings between one "style" of VHDL[1] to another "style" of VHDL. By the term "style", we mean a VHDL description containing VHDL constructs acceptable to a computer-aided design (CAD) system. Currently, vendor design tools are not sufficiently sophisticated to accept the entire VHDL language. As such, a vendor will specify a "style" of VHDL acceptable for their tool which is usually a subset of the VHDL language. This is the same for *vhdl2ges* (Dukes 91b).

A Prolog-parser called *vhdl_parser*[2] is used to generate a Prolog intermediate form of the VHDL model to be translated. The translated intermediate form is translated back to VHDL using the pretty-printer, **write_vhdl_design_units/4**, included in *vhdl_parser*. The Quintus[3] Prolog environment is used.

The Overall Translation Process There are three translations being performed. The first translation takes a VHDL description and generates a Prolog-intermediate form as defined by *vhdl_parser*. The second translation involves applying a mapping from one VHDL construct (as it is represented in the Prolog-intermediate form) to another VHDL construct (the goal of this project). In essence, a Prolog-intermediate form is generated from another Prolog-intermediate form. The final translation generates a VHDL description from the Prolog-intermediate form.

Assumptions There are a few assumptions concerning what is acceptable. The first assumption is that the VHDL provided to *vhdl_parser* is correct VHDL. One method of determining the correctness of the VHDL model is to have it analyzed through Zycad[4] VHDL. Another assump-

[1] VHDL as defined by the IEEE std 1076-1987 VHDL Language Reference Manual, hereafter called LRM.
[2] Copyright 1990 by the Microelectronics Center of North Carolina (Reint 90)
[3] Quintus and Quintus Prolog are trademarks of Quintus Computer Systems, Inc.
[4] Zycad VHDL is a trademark of Synopsis, Inc.

tion is that *vhdl_parser* works correctly. Oddly, some errors have been uncovered with *vhdl_parser*; however, we need this for a small "sanity check."

The purpose of this section is to present an overview of the Prolog-intermediate form used to represent VHDL models. The Prolog-intermediate form is generated from a VHDL model using *vhdl_parser*. The documentation accompanying the *vhdl_parser* did not include a description of the Prolog-intermediate form. Therefore, this chapter should be helpful to others trying to use *vhdl_parser*.

The basic Prolog representation for a VHDL model is **design_unit/2**. The first term of **design_unit/2** is a description of the library or package dependencies for the particular design unit under consideration. No detailed understanding of the first term is required, but is mentioned only for completeness.

The second term of **design_unit/2** is of great importance, since it holds the Prolog-intermediate form for a **package**, **package body**, **entity**, **configuration**, or **architecture**. The **configuration** is not directly related to this project.

The entity The format for an **entity** is the following.

```
design_unit(Use,entity(EntityName,Generics,PortMap,Declarations,EntityBody)).
```

The terms and types of **entity/5** are shown in Table 4. For the following VHDL model of an entity,

```
entity full_part is
  generic(constant tPLH : time);
  port(a,b : in bit;
       c   : out bit);
  attribute loc:integer;
  begin
    process (b)
      begin
      assert (a = '0');
      end process;
  end full_part;
```

the following Prolog-intermediate form results

```
design_unit(
  [],
  entity(
    full_part,
    [
      interface_element(constant,[tplh],null,
        vhdl_subtype(null,time,null),null,null)
    ],
    [
      interface_element(null,[a,b],in,vhdl_subtype(null,bit,null),null,null),
      interface_element(null,[c],out,vhdl_subtype(null,bit,null),null,null)
    ],
    [attribute_declaration(loc,integer)],
    [
      vhdl_process(
        null,
        [b],
        [],
        [assert(expr(a,=,char(48)),null,null)] )
    ] )).
```

Table 4. Terms and Types of **entity/5**

Terms	Types
EntityName	atom
Generics	list_of_interface_elements
PortMap	list_of_interface_elements
Declarations	list_of_declarative_objects
EntityBody	list_of_concurrent_statements

The architecture The format for an **architecture** is the following.

```
arch(ArchitectureName,EntityName,Delcarations,ArchitectureBody).
```

The terms and types of **architecture/4** are shown in Table 5. From the following VHDL model

```
architecture full_part of full_part is
  signal d : bit;
  begin
  c <= a and b;
  end full_part;
```

the following Prolog-intermediate form is generated.

```
design_unit(
  [],
  arch(
    full_part,
    full_part,
    [
      object_declaration(
        signal,[d],vhdl_subtype(null,bit,null),null,null)
    ],
    [
      csas(
        null,
        csa(
          c,
          null,
          null,
          [
            wave(
              [
                event(expr(a,and,b),null)
              ],
              null )
          ] ))
    ] )).
```

Table 5. Terms and Types of **architecture/4**

Terms	Types
ArchitectureName	atom
EntityName	atom
Declarations	list_of_declarative_objects
ArchitectureBody	list_of_concurrent_statements

The package The format for a **package** is the following.

```
package(PackageName,Delcarations).
```

The terms and types of **package/2** are shown in Table 6. From the following VHDL model

```
package Functions is
  type opcode is (oct0, oct1, oct2, oct3, oct4, oct5, oct6, oct7);
  function mnemonic (bit_pattern : in opcode) return integer;
  procedure mnemonic (bit_pattern : in opcode;answer : out integer);
  end Functions;
```

the following Prolog-intermediate form is generated.

```
design_unit(
  [],
  package(
    functions,
    [
      vhdl_type(opcode,[oct0,oct1,oct2,oct3,oct4,oct5,oct6,oct7]),
      sub_program(
        sub_spec(
          mnemonic,
          [
            interface_element(
              null,
              [bit_pattern],
              in,
              vhdl_subtype(null,opcode,null),null,null)
          ],
          integer ),
        null ),
      sub_program(
        sub_spec(
          mnemonic,
          [
            interface_element(
              null,
              [bit_pattern],in,
              vhdl_subtype(null,opcode,null),
              null,
              null ),
            interface_element(
              null,
              [answer],out,
              vhdl_subtype(null,integer,null),
              null,
              null )
          ],
          null ),
        null )
    ] )).
```

Table 6. Terms and Types of **package/2**

Terms	Types
PackageName	atom
Declarations	list_of_declarative_objects

The package body The format for a **package body** is the following.

```
package_body(PackageName,Delcarations).
```

The terms and types of **package_body/2** are shown in Table 7. From the following VHDL model

```
package body functions is

  function mnemonic (bit_pattern : in opcode) return integer is
    variable a,b,c : integer;  -- just for noise
    begin
    a := b + c;
    case bit_pattern is
      when oct0 => return(a);
      when oct1 => return(a+b);
      when oct2 => return(a+c);
      when oct3 => return(a+a);
      when oct4 => return(b+c);
      when oct5 => return(b+b);
      when oct6 => return(c+c);
      when oct7 => return(b);
      end case;
    end mnemonic;

  procedure mnemonic (bit_pattern : in opcode;answer : out integer) is

    variable a,b,c : integer;  -- just for noise
    begin
    a := b + c;
    case bit_pattern is
      when oct0 => answer := a;
      when oct1 => answer := a+b;
      when oct2 => answer := a+c;
      when oct3 => answer := a+a;
      when oct4 => answer := b+c;
      when oct5 => answer := b+b;
      when oct6 => answer := c+c;
      when oct7 => answer := b;
      end case;
    end mnemonic;

  end functions;
```

the following Prolog-intermediate form is generated.

```
design_unit(
  [],
  package_body(
    functions,
    [
      sub_program(
        sub_spec(
          mnemonic,
          [
            interface_element(
              null,
              [bit_pattern],in,
              vhdl_subtype(null,opcode,null),
              null,
              null )
          ],
          integer ),
        program_body(
          [
            object_declaration(
              variable,
              [a,b,c],
              vhdl_subtype(null,integer,null),
              null,
              null )
          ],
          [
            assign(a,expr(b,+,c)),
            case(
              bit_pattern,
              [
                vhdl_case([oct0],[return(a)]),
                vhdl_case([oct1],[return(expr(a,+,b))]),
                vhdl_case([oct2],[return(expr(a,+,c))]),
                vhdl_case([oct3],[return(expr(a,+,a))]),
                vhdl_case([oct4],[return(expr(b,+,c))]),
                vhdl_case([oct5],[return(expr(b,+,b))]),
                vhdl_case([oct6],[return(expr(c,+,c))]),
                vhdl_case([oct7],[return(b)])
              ] )
          ] )),
      sub_program(
        sub_spec(
          mnemonic,
          [
            interface_element(null,[bit_pattern],in,
              vhdl_subtype(null,opcode,null),null,null),
            interface_element(null,[answer],out,
              vhdl_subtype(null,integer,null),null,null)
          ],
          null ),
```

```
    program_body(
      [
        object_declaration(variable,[a,b,c],
          vhdl_subtype(null,integer,null),null,null)
      ],
      [
        assign(a,expr(b,+,c)),
        case(
          bit_pattern,
          [
            vhdl_case([oct0],[assign(answer,a)]),
            vhdl_case([oct1],[assign(answer,expr(a,+,b))]),
            vhdl_case([oct2],[assign(answer,expr(a,+,c))]),
            vhdl_case([oct3],[assign(answer,expr(a,+,a))]),
            vhdl_case([oct4],[assign(answer,expr(b,+,c))]),
            vhdl_case([oct5],[assign(answer,expr(b,+,b))]),
            vhdl_case([oct6],[assign(answer,expr(c,+,c))]),
            vhdl_case([oct7],[assign(answer,b)])
          ] )
      ] ))
] )).
```

Table 7. Terms and Types of **package_body/2**

Terms	Types
PackageName	atom
Declarations	list_of_declarative_objects

Translating Data Flow to Structure

Analysis of the VHDL Model Data-flow VHDL models express the composition of circuits at a gate level. Each data-flow statement is a digital-logic expression, representing a result in terms of the logical composition of its signals. An example of a data-flow statement is

```
architecture data_flow of example is
    signal a,b,c,d : bit;
    begin
    a <= b and c or d;
    end data_flow;
```

A data-flow statement also contains implicit properties. One such property is a signal connecting the result of **b and c** to the following **or** operator. An equivalent structural VHDL construction would look like

```
architecture structural of example is
    signal a,b,c,d : bit;
    signal internal_signal : bit;
    component and_gate
        port(a,b : in bit;
             c   : out bit)
        end component;
    component or_gate
        port(a,b : in bit;
             c   : out bit)
        end component;
    begin
    and_gate : and_gate
        port map(b,c,internal_signal);
    or_gate  : or_gate
        port map(internal_signal,d,a);
    end structural;
```

Another implicit property of the data-flow statement is that all logical connectives are binary operators except for the **NOT** operator. **NOT** is an unary operator. In consideration of the language operators, **AND**, **OR**, **NAND**, **NOR**, **XOR**, and **NOT**, we need only consider implicit signal declarations and six different components declarations. There are then two requirements for translating data-flow VHDL models to structural models. Implicit internal signals must be generated, declared, and placed. Secondly, the necessary components must be declared and instantiated with the correct interconnecting signals.

In order to analyze the Prolog-intermediate form generated for a full VHDL description of a data flow model, we will choose a description of a full adder, shown below.

```
entity full_adder is
    port (A,B,Cin  : in bit;
          Sum,Carry: out bit);
    end full_adder;

architecture data_flow of full_adder is

    signal interm1 : bit;

    begin

    interm1 <= a xor b;
    sum     <= interm1 xor cin;
    carry   <= (a and b) or (a and cin) or (b and cin);

    end data_flow;
```

The Prolog-intermediate form generated for the entity is the following.

```
design_unit(
  [],
  entity(
    full_adder,
    null,
    [
      interface_element(null,[a,b,cin],in,
        vhdl_subtype(null,bit,null),null,null),
      interface_element(null,[sum,carry],out,
        vhdl_subtype(null,bit,null),null,null)
    ],
    [],
    [] )).
```

The Prolog-intermediate form generated for the architecture is the following.

```
  design_unit(
    [],
    arch(
      data_flow,
      full_adder,
      [
        object_declaration(signal,[interm1],
          vhdl_subtype(null,bit,null),null,null)
      ],
      [
```

```
csas(
  null,
  csa(
    interm1,
    null,
    null,
    [
      wave(
        [
          event(
            expr(a,xor,b),
            null)
        ],
        null )
    ] )),
csas(
  null,
  csa(
    sum,
    null,
    null,
    [
      wave(
        [
          event(
            expr(interm1,xor,cin),
            null )
        ],
        null )
    ] )),
csas(
  null,
  csa(
    carry,
    null,
    null,
    [
      wave(
        [
          event(
            expr(
              expr(
                expr(a,and,b),
                or,
                expr(a,and,cin) ),
              or,
              expr(b,and,cin) ),
            null )
        ],
        null )
    ] ))
```

```
] )).
```

From the Prolog-intermediate form, we are interested in two items. The first is the explicit signal declarations in the entity and architecture. The second item is the expression trees formed by the concurrent signal assignment statements. Figure 27 shows the three expression trees formed by the three concurrent signal assignment statements from **architecture data_flow of full_adder**.

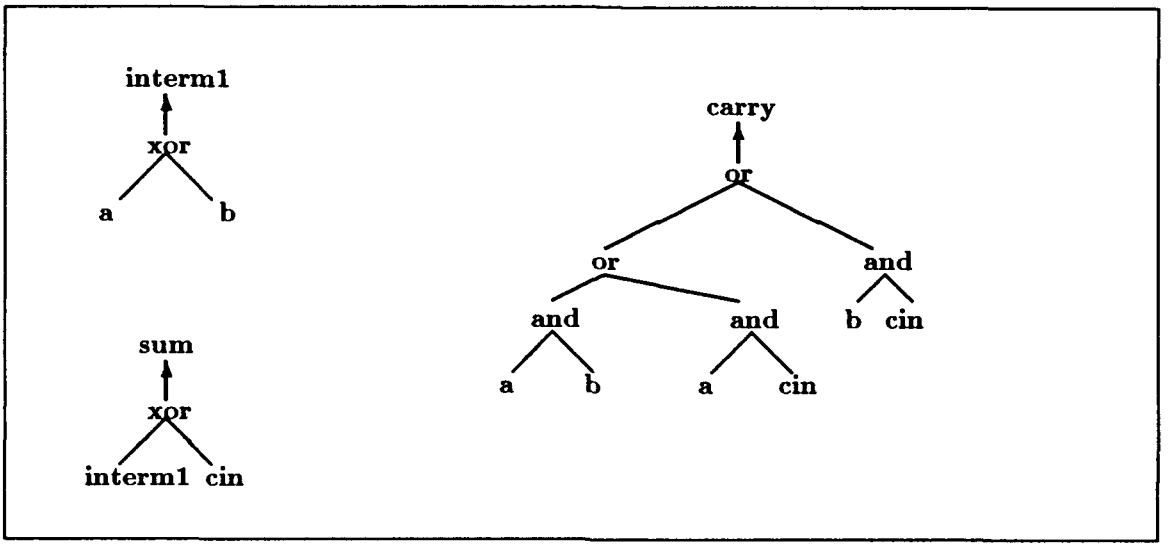


Figure 27. Expression Trees from Concurrent Signal Assignment Statements.

Generating Structural VHDL The Prolog written to translate from data-flow VHDL to structural VHDL generated the following result.

```
--  VHDL DESIGN UNIT #1
entity full_adder is
     port
        (a, b, cin:in bit;
        sum, carry:out bit);

end full_adder;

--  VHDL DESIGN UNIT #2
architecture sic_data_flow of full_adder is
```

```
component not_gate generic
    (constant tplh:time := 0 ns;
    constant tphl:time := 0 ns);
 port
    (a:in bit;
    b:out bit);
end component ;
for all : not_gate
use  entity work.inv(inv)
;
component xor_gate generic
    (constant tplh:time := 0 ns;
    constant tphl:time := 0 ns);
 port
    (a, b:in bit;
    c:out bit);
end component ;
for all : xor_gate
use  entity work.xor_gate(xor_gate)
;
component or_gate generic
    (constant tplh:time := 0 ns;
    constant tphl:time := 0 ns);
 port
    (a, b:in bit;
    c:out bit);
end component ;
for all : or_gate
use  entity work.or_gate(or_gate)
;
component and_gate generic
    (constant tplh:time := 0 ns;
    constant tphl:time := 0 ns);
 port
    (a, b:in bit;
    c:out bit);
end component ;
for all : and_gate
use  entity work.and_gate(and_gate)
;
signal map_d2s0:bit;
signal map_d2s1:bit;
signal map_d2s2:bit;
signal map_d2s3:bit;
signal map_d2s4:bit;
signal interm1:bit;
begin
    xor_gate0 : xor_gate generic map
        (0 ns, 0 ns)
     port map
        (a, map_d2s0, interm1)
```

```
        ;
        not_gate0 : not_gate generic map
            (0 ns, 0 ns)
         port map
            (b, map_d2s0)
        ;
        xor_gate1 : xor_gate generic map
            (0 ns, 0 ns)
         port map
            (interm1, cin, sum)
        ;
        or_gate0 : or_gate generic map
            (0 ns, 0 ns)
         port map
            (map_d2s1, map_d2s2, carry)
        ;
        or_gate1 : or_gate generic map
            (0 ns, 0 ns)
         port map
            (map_d2s3, map_d2s4, map_d2s1)
        ;
        and_gate0 : and_gate generic map
            (0 ns, 0 ns)
         port map
            (a, b, map_d2s3)
        ;
        and_gate1 : and_gate generic map
            (0 ns, 0 ns)
         port map
            (a, cin, map_d2s4)
        ;
        and_gate2 : and_gate generic map
            (0 ns, 0 ns)
         port map
            (b, cin, map_d2s2)
        ;

end sic_data_flow;
```

From the structural VHDL code, the following may be noticed. The correct component declarations were made only for those components to be instantiated. Declaring components that are not instantiated later can cause errors for some VHDL design systems. The correct signals were declared for the implicit signals in the data-flow model. Lastly, all of the components were instantiated for those operators in the data-flow model.

Limitations and Features Currently, the translator does not handle signals declared through the **alias** keyword. Furthermore, the translator has not been adapted to consider **bit_vector** signals. Other nondata-flow constructs in the architectural body of the VHDL model under consideration are not touched. Therefore, sequential data-flow statements within a **process** are not translated.

Since a VHDL model may have component instantiations mixed with data-flow statements, passing nondata-flow language constructs through untouched would yield the benefit of deriving a fully structural model from a mixed data-flow/structural VHDL model. Handling **bit_vectors** was also not considered in this step since it could be handled by another translation step. Therefore, the translation from data-flow to structure could be kept simple.

Running *d2s*

d2s is tested through the use of the *make* utility in much the same manner as *case2if*. This is to help reduce the number of keystrokes necessary to accomplish the task of building *d2s* and testing it. Dependencies are set up for all test cases so that if *d2s* has not been built, it will be automatically before testing. The expected results are kept in a directory called data. Thus, the Unix *diff* utility may be used to compare the actual output of *d2s* with the expected in each test case.

The two test cases were derived in this fashion. *d2s* was written first. The VHDL models generated by *d2s* were then analyzed and simulated for comparison against the original model. The two results are placed in the directory called data.

The procedure for logging into VERIFY.EL.WPAFB.AF.MIL and running the test cases is the same as for *case2if*. The two test cases are explained below.

c1data This test case is a VHDL model of a full adder described using concurrent signal assignment statements.

form7 This test case is a model of a seven-input parity generator. The components were arranged in such a fashion to provide as large a concurrent signal assignment statement as possible. Included in this VHDL model is a **process** statement.

Prolog Code for d2s

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    d2s/1 is a Prolog routine to convert a VHDL model
%        with data flow constructs to an equivalent VHDL model
%        with component instantiations.  d2s/1 calls upon vhdl_read/1
%        of vhdl_parser to parse the original VHDL model.
%        The expression trees formed by the VHDL concurrent
%        signal assignment statements are used to generate the
%        necessary implicit signal declarations as well as
%        components.  Finally, the new VHDL model is generated through
%        write_vhdl_design_units/4 using a pretty-printer
%        supplied in vhdl_parser.
%
%    Limitation:  In order to ensure all signal declarations are
%        available for use, an entity with its respective
%        architecture must exist in the one VHDL file supplied.
%
%  d2s/1 must be loaded into vhdl_parser!  The command is
%
%  % vhdl_parser
%
%  Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
%  Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
%  1310 Villa Street, Mountain View, California  (415) 965-7700
%
%  | ?- compile(d2s).
%
%  Afterwards, save the executable image in the following manner...
%
%  | ?- save(d2s).
%
%  Execute by the following:
%
%  | ?- d2s(foo).
%
%
%  Several tables are built in memory.
%
%    signal_name(Name).                  So we don't have duplicate
%                                        signal names.
%
%    expression_tree(ResultName,Tree).   Where the data_flow statements
%                                        are stored.
%
%    comp_table(Name,Num).               Components to be declared.
%                                        And number instantiated.
%
```

```
%    new_signal_name_num(Num).          Number of how many signal
%                                       names have been generated.
%    new_signal_name(Name).             Name of new signal


d2s(File) :-
    vhdl_read(File,DesignUnits),
    map_d2s(DesignUnits,NewDesignUnits),
    tell('outfile.vhd'),
    write_vhdl_design_units(NewDesignUnits,0,L,[]),
    write_list(L,0),
    told.




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    test/2 was supplied for testing purposes.

test(
    [design_unit(Use,entity(EntityName,Generic,Port,EntityDecl,EntityBody)),
     design_unit(Use,arch(ArchName,EntityName,ArchDecl,ArchBody))],
    [design_unit(Use,entity(EntityName,Generic,Port,EntityDecl,EntityBody)),
     design_unit(Use,arch(ArchName,EntityName,NewArchDecl,NewArchBody))]) :-
        !,
        map_d2s_build_signal_name_table(Port),
        map_d2s_build_signal_name_table(ArchDecl),
        map_d2s_build_expression_tree_table(ArchBody,NewArchBody),
        map_d2s_generate_new_signals(Signals),
        append(Signals,ArchDecl,NewArchDecl).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s/2 main driver routine for breaking out the signal
%        names and expression tree with the VHDL entity and
%        architecture pair.  Signal tables are built from
%        the entity and architecture.  Components are contructed
%        from the expression trees formed by the concurrent
%        signal assignment statements.  If the one entity
%        and architecture rule is not adhered to, a warning
%        is issued.

map_d2s(
    [design_unit(UseE,entity(EntityName,Generic,Port,EntityDecl,EntityBody)),
     design_unit(UseA,arch(ArchName,EntityName,ArchDecl,ArchBody))],
    [design_unit(UseE,entity(EntityName,Generic,Port,EntityDecl,EntityBody)),
     design_unit(UseA,arch(ArchName,EntityName,NewArchDecl,NewArchBody))]) :-
        !,
        map_d2s_build_signal_name_table(Port),
        map_d2s_build_signal_name_table(ArchDecl),
        map_d2s_build_expression_tree_table(ArchBody,NewArchBody),
        map_d2s_generate_new_signals(Signals),
        map_d2s_generate_new_comps(Comps),
        append(Signals,ArchDecl,IntermArchDecl),
        append(Comps,IntermArchDecl,NewArchDecl).
map_d2s(_,_) :-
        write('Please place one entity and its associated'),nl,
        write('architecture in one file before rerunning'),nl,
        write('d2s/1.'),nl,
        fail.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_generate_new_comps/1 is used for building component
%        declaration in the declarative region of the architecture.

map_d2s_generate_new_comps([
      vhdl_comp(
        not_gate,
        [
          interface_element(constant,[tplh],null,
            vhdl_subtype(null,time,null),null,vhdl_assign(pl(0,ns))),
          interface_element(constant,[tphl],null,
            vhdl_subtype(null,time,null),null,vhdl_assign(pl(0,ns)))
        ],
        [
          interface_element(null,[a],in,
            vhdl_subtype(null,bit,null),null,null),
          interface_element(null,[b],out,
            vhdl_subtype(null,bit,null),null,null)
        ] ),
      vhdl_spec(spec(all,not_gate),binding(entity_aspect(
            vhdl_name(prefix(work),inv),
            inv ),
          null,
          null )) | Comps]) :-
        retract(comp_table(not,_Num)),
        !,
        map_d2s_generate_new_comps(Comps).
map_d2s_generate_new_comps([
      vhdl_comp(
        GateName,
        [
          interface_element(constant,[tplh],null,
            vhdl_subtype(null,time,null),null,vhdl_assign(pl(0,ns))),
          interface_element(constant,[tphl],null,
            vhdl_subtype(null,time,null),null,vhdl_assign(pl(0,ns)))
        ],
        [
          interface_element(null,[a,b],in,
            vhdl_subtype(null,bit,null),null,null),
          interface_element(null,[c],out,
            vhdl_subtype(null,bit,null),null,null)
        ] ),
      vhdl_spec(spec(all,GateName),binding(entity_aspect(
            vhdl_name(prefix(work),GateName),
            GateName ),
          null,
          null )) | Comps]) :-
        retract(comp_table(Name,_Num)),
        !,
```

```
        name('_gate',Suffix),
        name(Name,Prefix),
        append(Prefix,Suffix,GateNameList),
        name(GateName,GateNameList),
        map_d2s_generate_new_comps(Comps).
map_d2s_generate_new_comps([]).




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_generate_new_signals/1 is used for generating a list
%        of signal declarations for the newly formed signals.
%        The signal declarations are placed in the delcarative
%        region of the architecture.

map_d2s_generate_new_signals(
    [object_declaration(signal,[Name],
        vhdl_subtype(null,bit,null),null,null)|ObjDeclList]) :-
        retract(new_signal_name(Name)),
        !,
        map_d2s_generate_new_signals(ObjDeclList).
map_d2s_generate_new_signals([]).

map_d2s_build_signal_name_table(
    [interface_element(_sig,SigList,_mode,
        vhdl_subtype(null,bit,null),null,null)|SignalNames]) :-
        map_d2s_build_signal_name_list_table(SigList),
        !,
        map_d2s_build_signal_name_table(SignalNames).
map_d2s_build_signal_name_table(
    [object_declaration(_sig,SigList,
        vhdl_subtype(null,bit,null),null,null)|SignalNames]) :-
        map_d2s_build_signal_name_list_table(SigList),
        !,
        map_d2s_build_signal_name_table(SignalNames).
map_d2s_build_signal_name_table([_DeclItem|SignalNames]) :-
        !,
        map_d2s_build_signal_name_table(SignalNames).
map_d2s_build_signal_name_table([]).
map_d2s_build_signal_name_table(null).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_build_signal_name_list_table/1 builds a table of
%        declared signals.  The table is used to keep track
%        of signals that are already declared.

map_d2s_build_signal_name_list_table([Signal|SigList]) :-
        assert(signal_name(Signal)),
        !,
        map_d2s_build_signal_name_list_table(SigList).
map_d2s_build_signal_name_list_table([]).




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_build_expression_tree_table/2 builds a list of
%        component instantiations to be placed in the architecture
%        body.

map_d2s_build_expression_tree_table(
    [csas(Trans,Csa)|ArchBody],NewArchBody) :-
        map_d2s_convert_csas_to_comp(csas(Trans,Csa),CompInst),
        !,
        map_d2s_build_expression_tree_table(ArchBody,InterArchBody),
        append(CompInst,InterArchBody,NewArchBody).
map_d2s_build_expression_tree_table(
    [Head|ArchBody],[Head|NewArchBody]) :-
        !,
        map_d2s_build_expression_tree_table(ArchBody,NewArchBody).
map_d2s_build_expression_tree_table([],[]).




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_convert_csas_to_comp/2 is used for breaking down
%        an expression tree.

map_d2s_convert_csas_to_comp(
    csas(null,csa(SignalName,null,null,[wave([event(Expr,null)],null)])),
    CompInstList) :-
    map_d2s_convert_expr_to_comp(SignalName,Expr,CompInstList).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_convert_expr_to_comp/2 is where the component is
%        derived from an expression.

map_d2s_convert_expr_to_comp(
    SignalName,
    expr(Opr,SignalNameR),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameR),
         element(null,SignalName)])] ) :-
        atom(SignalNameR),!,
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
        !.
map_d2s_convert_expr_to_comp(
    SignalName,
    expr(Opr,Expr),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameR),
         element(null,SignalName)])|CompListR] ) :-
        !,
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
        map_d2s_gen_signal_name(SignalNameR),
        map_d2s_convert_expr_to_comp(
            SignalNameR,
            Expr,
            CompListR),
        !.
map_d2s_convert_expr_to_comp(
    SignalName,
    expr(SignalNameL,Opr,SignalNameR),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameL),element(null,SignalNameR),
         element(null,SignalName)])] ) :-
        atom(SignalNameL),
        atom(SignalNameR),!,
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
        !.
map_d2s_convert_expr_to_comp(
    SignalName,
    expr(Expr,Opr,SignalNameR),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameL),element(null,SignalNameR),
         element(null,SignalName)])|CompListL] ) :-
        atom(SignalNameR),!,
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
```

```
        map_d2s_gen_signal_name(SignalNameL),
        map_d2s_convert_expr_to_comp(
            SignalNameL,
            Expr,
            CompListL),
        !.
map_d2s_convert_expr_to_comp(
    SignalName,
    expr(SignalNameL,Opr,Expr),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameL),element(null,SignalNameR),
         element(null,SignalName)])|CompListR] ) :-
        atom(SignalNameL),!,
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
        map_d2s_gen_signal_name(SignalNameR),
        map_d2s_convert_expr_to_comp(
            SignalNameR,
            Expr,
            CompListR),
        !.
map_d2s_convert_expr_to_comp(
    SignalName,
    expr(ExprL,Opr,ExprR),
    [comp_instant(CompInstName,CompName,
        [element(null,pl(0,ns)),element(null,pl(0,ns))],
        [element(null,SignalNameL),element(null,SignalNameR),
         element(null,SignalName)])|CompList] ) :-
        map_d2s_get_comp_name_inst(Opr,CompInstName,CompName),
        map_d2s_gen_signal_name(SignalNameL),
        map_d2s_gen_signal_name(SignalNameR),
        map_d2s_convert_expr_to_comp(
            SignalNameL,
            ExprL,
            CompListL),
        map_d2s_convert_expr_to_comp(
            SignalNameR,
            ExprR,
            CompListR),
        append(CompListL,CompListR,CompList),
        !.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_gen_signal_name/1 builds a table of newly created
%        signals.  The table is to be used for reconstructing
%        the declarative region of the architecture later.

map_d2s_gen_signal_name(SignalName) :-
        retract(new_signal_name_num(Num)),
        !,
        name(Num,NumList),
        name(map_d2s,NameList),
        append(NameList,NumList,SigNameList),
        name(TmpSigName,SigNameList),
        map_d2s_return_good_name(Num,TmpSigName,IntNum,SignalName),
        NewNum is IntNum + 1,
        assert(new_signal_name(SignalName)),
        assert(new_signal_name_num(NewNum)),!.
map_d2s_gen_signal_name(SignalName) :-
        name(0,NumList),
        name(map_d2s,NameList),
        append(NameList,NumList,SigNameList),
        name(TmpSigName,SigNameList),
        map_d2s_return_good_name(0,TmpSigName,IntNum,SignalName),
        NewNum is IntNum + 1,
        assert(new_signal_name(SignalName)),
        assert(new_signal_name_num(NewNum)),!.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_return_good_name/4 is used to ensure that a newly
%        created signal name doesn't already exist.

map_d2s_return_good_name(Num,TmpSigName,NewNum,SignalName) :-
        signal_name(TmpSigName),
        !,
        IntNum is Num + 1,
        name(map_d2s,NameList),
        name(IntNum,IntNumList),
        append(NameList,IntNumList,IntSigNameList),
        name(IntSigName,IntSigNameList),
        !,
        map_d2s_return_good_name(IntNum,IntSigName,NewNum,SignalName).
map_d2s_return_good_name(Num,SigName,Num,SigName).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    map_d2s_get_comp_name_inst/3 is used to generated component
%        labels in the archtecture body.

map_d2s_get_comp_name_inst(Opr,CompInstName,CompName) :-
        retract(comp_table(Opr,Num)),
        !,
        name(Opr,GateName),
        name('_gate',Extension),    %I know this looks inefficient
                                    %but I want this to work on
                                    %ANY machine.
        append(GateName,Extension,CompNameList),
        name(CompName,CompNameList),
        name(Num,NumList),
        append(CompNameList,NumList,CompInstNameList),
        name(CompInstName,CompInstNameList),
        NewNum is Num + 1,
        assert(comp_table(Opr,NewNum)),!.
map_d2s_get_comp_name_inst(Opr,CompInstName,CompName) :-
        name(Opr,GateName),
        name('_gate',Extension),    %I know this looks inefficient
                                    %but I want this to work on
                                    %ANY machine.
        append(GateName,Extension,CompNameList),
        name(CompName,CompNameList),
        name(0,Num),
        append(CompNameList,Num,CompInstNameList),
        name(CompInstName,CompInstNameList),
        assert(comp_table(Opr,1)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    UTILITIES
%

%append([],L,L).
%append([H|L1],L2,[H|L3]) :-
%  append(L1,L2,L3).
```

Corrections to *vhdl_parser*

The first correction made to *vhdl_parser* involved the proper differentiation between **package** and **package body**. Originally, *vhdl_parser* would translate the following VHDL

```
package body functions is

  function mnemonic (bit_pattern : in opcode) return integer is
    variable c : integer;  -- just for noise
    begin
    return(c);
    end mnemonic;

  end functions;
```

to

```
--  VHDL DESIGN UNIT #1
package functions is
    function mnemonic(bit_pattern:in opcode) return integer is
        variable c:integer;

    begin
         return c;

    end mnemonic;

end functions;
```

which produced an invalid VHDL **package**.

The corrections made to *vhdl_parser* are shown below.

In the file **vhdl.tex**, Rule 16 was changed from

```
vhdl_package_body(package(ID,DIs)) -->
       [ package,  body ], vhdl_identifier(ID), [ is ],
         vhdl_opt_declarative_items(DIs),
       [ end ], vhdl_opt_identifier(ID).
```

to

```
vhdl_package_body(package_body(ID,DIs)) -->
       [ package,  body ], vhdl_identifier(ID), [ is ],
         vhdl_opt_declarative_items(DIs),
       [ end ], vhdl_opt_identifier(ID).
```

In the file **vhdl_write.tex**, Rule 16 was changed from

```
write_vhdl_package_body(package(ID,DIs)) -->
       "package  body ", write_vhdl_identifier(ID), " is ",
     [indent],
         write_vhdl_opt_declarative_items(DIs),
     [undent],
       "end ", write_vhdl_opt_identifier(ID).
```

to

```
write_vhdl_package_body(package_body(ID,DIs)) -->
       "package  body ", write_vhdl_identifier(ID), " is ",
     [indent],
         write_vhdl_opt_declarative_items(DIs),
     [undent],
       "end ", write_vhdl_opt_identifier(ID).
```

Another problem encountered with *vhdl_parser* was the pretty-printing of physical types. For a line that looks like

```
6 ns;
```

the pretty-printed result would look like

```
6ns;
```

leaving a syntactically-incorrect VHDL model. In order to correct this problem, Rule 5 in **vhdl_write.tex** was changed from

```
write_vhdl_physical_literal(pl(AL,ID)) -->
        write_vhdl_abstract_literal(AL),
        write_vhdl_identifier(ID).
```

to

```
write_vhdl_physical_literal(pl(AL,ID)) -->
        write_vhdl_abstract_literal(AL)," ",
        write_vhdl_identifier(ID).
```

Uncorrected Problems with *vhdl_parser*

Listed in this appendix are errors encountered in *vhdl_parser* that have not been corrected. Both errors involve subtleties with special characters and integers.

The first error involves integer representation in *vhdl_parser* when a **bit_vector** is intended. Leading zeros in **bit_vectors** are dropped due to the conversion to integer during file input with *vhdl_get_token_line/1*. For an input line with **001** as a **bit_vector**, the resulting representation is **1**. The pretty-printed VHDL code will contain a syntax error due to this problem.

The second error involves the use of the quote character, ; in VHDL. The quote character is essentially dropped. Therefore, an **assert** statement,

```
assert (expected = actual) report "conflict" severity error;
```

is reprinted as

```
assert (expected = actual)
report conflictseverity error;
```

causing a syntax error.

Conclusion

The process of translating data-flow to structure was successful in that the resulting VHDL code yielded the simulation results as the original VHDL code. Separating the *d2s* from *ges* allowed for ease of testing, isolation from changes, and ease of code development. Also worth noting is that

a partial data-flow VHDL model will only have the data-flow portion changed by *d2s*, leaving the rest of the VHDL model alone.

The errors found in *vhdl_parser* were noted and fixed as indicated. Not all errors were corrected. The errors remaining to be corrected were not devastating to tool development.

Appendix E. Formal Methods

Formal methods are used to provide a systematic basis for specifying, developing, and verifying relations between a specification and an implementation (Wing 90b:8). Some of the relations or properties examined by formal methods may include, but are not limited to, equivalence, implication, reliability, safety, liveness, consistency, or completeness. Some formal methods familiar to the design engineer involved in VLSI design include design-rule checking, synthesis, silicon compilation, and petri nets. These formal methods have a mathematical basis; however, methods that involve *ad hoc* simulation are not considered formal.

Logic extraction is a formal method that is used to verify the equivalence between a component netlist and a VHDL structural description. The VHDL description is the structural specification and the component netlist represents the layout specification[1]. Furthermore, logic extraction may also be used to examine configuration properties (i.e., design rule checks), reliability properties, and temporal properties of a digital circuit.

To help show that logic extraction is a formal method, we will use the notation

$$\Gamma \vdash \alpha$$

(Duffy 91:31-34,43-54) where Γ is a set of assumptions or axioms and α is the theorem to be proved. Γ sets the context for proving the theorem α. Another way of expressing $\Gamma \vdash \alpha$ is

$$\gamma_1 \wedge \ldots \wedge \gamma_n \Rightarrow \alpha$$

where each $\gamma_i \in \Gamma$ and $1 \leq i \leq n$.

[1] A transistor netlist is extracted from the layout specification using already-existing CAD tools.

Various rules of inference may be used to demonstrate that α follows from Γ. One inference rule is the rewrite rule. With respect to $\Gamma \vdash \alpha$, a rewrite rule applies a replacement in α specified by some γ_i. The replacement is not necessarily always a reducing one. For γ_i to be used as the basis for a rewrite, γ_i must be in the form $t = t'$. A replacement occurs through a rewrite in α when some expression of α is in the form of t[2]. For the formula

$$(C = A \wedge B) \vdash C \Rightarrow A \vee B$$

a rewrite would result in

$$(C = A \wedge B) \vdash (A \wedge B) \Rightarrow (A \vee B).$$

By the definition of $\Gamma \vdash \alpha$ the formula may be rewritten as

$$(C = A \wedge B), A, B \vdash A \vee B$$

which yields[3]

$$(C = A \wedge B), A, B \vdash \top \vee B$$

and finally

$$(C = A \wedge B), A, B \vdash \top.$$

In a logical sense, logic extraction may be viewed as

$$\textit{Extraction Rules} \vdash \textit{Layout Specification} \Leftrightarrow \textit{Structural Specification}$$

[2]The choice of matching t and replacing with t' versus matching t' and replacing with t is inconsequential provided the matching is done to prevent an infinite matching/replace cycle.

[3]$\top$ is used to denote logical true. Each γ_i is assumed true; therefore, everywhere that an expression in α matches a γ_i, $\top$ is substituted for the expression.

where *ExtractionRules* form the context Γ of α, the *Layout Specification* is represented by a component netlist, and the *StructuralSpecification* is the top-level component represented by the specification. Further *Layout Specification* $\Leftrightarrow$ *Structural Specification* is the theorem α.

As an example, consider the following.

$$(c_1(X,Y) \wedge c_2(X,Y) = c_3(X,Y)),\ (c_3(X,Y) \wedge c_4(Y,Z) = c_5(X,Y,Z)) \vdash (c_1(a,b) \wedge c_2(a,b) \wedge c_4(b,c)) \Leftrightarrow c_5(a,b,c) \quad (19)$$

$$(c_1(X,Y) \wedge c_2(X,Y) = c_3(X,Y)),\ (c_3(X,Y) \wedge c_4(Y,Z) = c_5(X,Y,Z)) \vdash (c_3(a,b) \wedge c_4(b,c)) \Leftrightarrow c_5(a,b,c) \quad (20)$$

$$(c_1(X,Y) \wedge c_2(X,Y) = c_3(X,Y)),\ (c_3(X,Y) \wedge c_4(Y,Z) = c_5(X,Y,Z)) \vdash c_5(a,b,c) \Leftrightarrow c_5(a,b,c) \quad (21)$$

In Eq 19, the *ExtractionRules* are listed as the assumptions concerning the environment of the proof. Also included is a list of the components from the *LayoutSpecification*. In Eq 20, a rewrite of the *LayoutSpecification* has been applied from the assumption list. The derivation in Eq 20 gives a new representation of the *LayoutSpecification*, but not in a form readily provable. Finally, in Eq 21, another rewrite of the *LayoutSpecification* is performed from the list of assumptions. The final result is readily proven true.

A discussion on the basis of formal methods may be found in (Wing 89), (Wing 90a), and (Wing 90b). Further information on formal methods may also be found in (MacLe 90:52-61), (Beth 62), (Ramsa 91), (Schar 88), and (Gordo 88).

Bibliography

[Barr 81] Barr, A. and E. A. Feigenbaum. *The Handbook of Artificial Intelligence, Vol. 1.* Los Altos: William Kaufman, Incorporated, 1981.

[Barro 84] Barrow, Harry G. "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence, 24:* 437-491 (1984).

[Beth 62] Beth, Evert W. *Formal Methods.* New York: Gordon and Breach Science Publishers, Incorporated, 1962.

[Boehn 88] Boehner, M. "LOGEX - An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology," *Proceedings of the IEEE/ACM Design Automation Conference.* 517-522. New York: IEEE Press, 1988.

[Boole 54] Boole, George. *An Investigation of the Laws of Thought.* Originally published in 1854 by Macmillan, London. Reprinted by Dover Publications in 1958.

[Boyer 79] Boyer, Robert S. and J. Strother Moore. *A Computational Logic,* Orlando: Academic Press, 1979.

[Bratk 86] Bratko, Ivan. *Prolog Programming for Artificial Intelligence,* Reading: Addison-Wesley Publishing Company, 1986.

[Brown 90] Brown, Frank Markham. *Boolean Reasoning.* Boston: Kluwer Academic Publishers, 1990.

[Bryan 87] Bryant, Randal E. "A Methodology for Hardware Verification Based on Logic Simulation," Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15217, ARPA Order Number 4976, (8 June 1987).

[Calif 86] California, University of, at Berkeley. Berkeley Distribution of Design Tools. Computer Science Division, EECS Department, University of California at Berkeley, 1986.

[Camil 86] Camilleri, Albert, Mike Gordon, and Tom Melham. *Hardware Verification Using Higher-Order Logic.* Technical Report No. 91, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, England, September 1986.

[Camur 88] Camurati, Paolo and Paolo Prinetto. "Formal Hardware Verification of Hardware Correctness: Introduction and Survey of Current Research," *Computer, 21:7* 8-19 (June 1988).

[Clock 87a] Clocksin, W. F. "Logic programming and Digital Circuit Analysis," *The Journal of Logic Programming, 4:1* 59-82 (March 1987).

[Clock 87b] Clocksin, W. F. and C. S. Mellish. *Programming in Prolog.* New York: Springer-Verlag, 1987.

[Cousi 86] Cousineau, G., G. Huet and L. Paulson. *The ML Handbook.* INRIA, 1986.

[deGeu 89] de Geus, Aart J. "Logic Synthesis Speeds ASIC Design," *IEEE Spectrum, 26:8* 27-31 (August 1989).

[Devad 88] Devadas, Srinivas, Hi-Kueng Tony Ma, and Alberto Sangiovanni-Vincentelli. "Logic Verification, Testing and their Relationship to Logic Synthesis," *Testing and Diagnosis of VLSI and ULSI* F. Lombardi and M. Sami, eds., Kluwer Academic Publishers, 181-245 (1988).

[Donne 68] Donnellan, Thomas. *Lattice Theory.* Oxford: Pergammon Press, Ltd., 1968.

[Duffy 91] Duffy, David A. *Principles of Automated Theorem Proving.* Chister: John Wiley and Sons Ltd., 1991.

[Dukes 88] Dukes, Captain Michael A. *A Multiple-Valued Logic System for Circuit Extraction to VHDL 1076-1987.* MS thesis, AFIT/GE/ENG/88S-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1988 (AD-A202646).

[Dukes 90a] Dukes, Michael Alan and Frank Markham Brown. *Proving Boolean Equivalence with Prolog.* January 1989-February1990. WRDC Technical Report, WRDC-TR-90-5006 Wright Research and Development Center, Wright-Patterson AFB OH, February 1990.

[Dukes 90b] Dukes, M. A., F. M. Brown, and J. E. DeGroat. "A Generalized Extraction System for VLSI," *VHDL Methods Workshop.* Center for Semicustom Integrated Systems at the University of Virginia and Design Automation Technical Committee of the IEEE Computer Society; 13-15 August 1990.

[Dukes 90c] Dukes, M. A., F. M. Brown, and J. E. DeGroat. *A Generalized Extraction System for VLSI.* July 1987-August 1990. WRDC Technical Report, WRDC-TR-90-5021, Wright Research and Development Center, Wright-Patterson AFB OH, August 1990.

[Dukes 91a] Dukes, M. A., F. M. Brown, and J. E. DeGroat. "Verification of Layout Descriptions Using GES," *Proceedings of the VHDL User's Group Spring 1991 Conference.* 63-72. Menlo Park: Conference Management Services; 8-10 April 1991.

[Dukes 91b] Dukes, M. A., F. M. Brown, and J. E. DeGroat. *A Prolog System for Converting VHDL-Based Models to Generalized Extraction System (GES) Rules.* June 1990 to December 1990. WL Technical Report, WL-TR-91-5018, Wright Laboratory, Wright-Patterson AFB OH, June 1991.

[Dukes 91c] Dukes, M. A., F. M. Brown, and J. E. DeGroat. "A Generalized Extraction System for VHDL," *Proceedings of the Fourth Annual IEEE International ASIC Conference and Exhibit.* P4-8.1-P4-8.4. New York: IEEE Press; 23-27 September 1991.

[Ebeli 83] Ebeling, C. and O. Zajicek. "Validating VLSI Circuit Layout by Wirelist Comparison," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 172-173. New York: IEEE Press; 1983.

[EIA 89] EIA VHDL Model Standards Committee, "EIA Commercial Component Model Specification (Preliminary)," 5 September 1989.

[Fujiw 85] Fujiwara, Hideo. *Logic Testing and Design for Testability.* Cambridge: The MIT Press, 1985.

[Galto 87] Galton, Antony, ed. *Temporal Logics and Their Applications.* San Diego: Academic Press, 1987.

[Gordo 87] Gordon, Michael. *The HOL Manual.* 1987.

[Gordo 88] Gordon, Michael. "HOL: A Proof Generating System for Higher-Order Logic," *VLSI Specification, Verification, and Synthesis,* 73-128. Boston: Kluwer Academic Publishers, 1988.

[Gordo 89] Gordon, Michael. *The HOL System Tutorial.* Cambridge Research Center of SRI International under a grant from DSTO Australia, 8 December 1989.

[Gupta 91] Gupta, Aarti. "Formal Hardware Verification Methods: A Survey," CMU Technical Report, CMU-CS-91-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15217, (October 1991).

[Hayes 88] Hayes, John P., *Computer Architecture and Organization*, New York: McGraw-Hill, Incoporated, 1988.

[Hehne 84] Hehner, Eric C. R., *The Logic of Programming*, Englewood Cliffs: Prentice Hall, Incorporated, 1984.

[IEEE 87] IEEE, Computer Society Standards Committee, "IEEE Standard VHDL Language Reference Manual," *ANSI/IEEE Std 1076-1987*, IEEE Press, New York, 1987.

[Linde 88] Linderman, R., K. Jones, and D. Gallagher, Sim-to-VHDL-Extractor (STOVE), computer program developed at the Air Force Institute of Technology, 1985-1988.

[MacLe 90] MacLennan, Bruce J. *Functional Programming: Practice and Theory.* Reading: Addison-Wesley Publishing Company, Incorporated, 1990.

[Mano 79] Mano, M. M. *Digital Logic and Computer Design.* Englewood Cliffs: Prentice-Hall, Incorporated, 1979.

[Mano 82] Mano, M. M. *Computer System Architecture,* Englewood Cliffs: Prentice-Hall, Incorporated, 1982.

[McClu 56] McCluskey, E.J., Jr. "Minimization of Boolean Functions," *Bell System Technical Journal, 35:* 1417-1444 (November 1956).

[Moszk 86] Moszkowski, Ben. *Executing Temporal Logic Programs.* New York: Cambridge, 1986.

[Ouste 84] Ousterhout, John K., "Switch-Level Delay Models for Digital MOS VLSI," *Proceedings of the ACM/IEEE 21st Design Automation Conference.* 542-548. New York: IEEE Press; 1984.

[Papas 88] Papaspyridis, A. C. "A Prolog-Based Connectivity Verification Tool," *Proceedings of the IEEE/ACM Design Automation Conference.* 523-527. New York: IEEE Press; 1988.

[Quint 88] Quintus Computer Systems, Incorporated, *Quintus Prolog Reference Manual,* Mountain View, California, 1988.

[Ramsa 91] Ramsay, Allan. *Formal Methods in Artificial Intelligence.* Cambridge: Cambridge University Press, 1991.

[Reint 90] Reintjes, P. B. "A VHDL Parser in Prolog," Technical Report, Research Triangle Park, North Carolina, 1990.

[Resch 69] Rescher, Nicholas. *Many-Valued Logic.* New York: McGraw-Hill, Incoporated, 1969.

[Rudea 74] Rudeanu, Sergiu. *Boolean Functions and Equations.* London: North-Holland Publishing, 1974.

[Schar 88] Scharbach, P. N., ed. *Formal Methods: Theory and Practice.* Boca Raton: CRC Press, Incoporated, 1988.

[Seraf 90a] Serafino, K. M. and M. A. Dukes. *VHSIC Hardware Description Language (VHDL) Benchmark Suite.* November 1989 to October 1990. WRDC Technical Report, WRDC-TR-90-5026, Wright Research and Development Center, Wright-Patterson AFB OH, October 1990.

[Seraf 90b] Serafino, K. M. and M. A. Dukes. *1990 VHDL Users' Group Fall Meeting.* 193-201. Menlo Park: Conference Management Services; 14-17 October 1990.

[Seraf 91a] Serafino, K. M. and M. A. Dukes. *VHDL and WAVES Descriptions for a Pseudo-Random Pattern Generator.* April 1991 to October 1991. WL Technical Report, WL-TR-91-5037, Wright Laboratory, Wright-Patterson AFB OH, June 1991.

[Seraf 91b] Serafino, K. M. and M. A. Dukes. VHDL and WAVES Descriptions for a Pseudo-Random Pattern Generator," *VHDL International Users' Forum.* 67-75. Menlo Park: Conference Management Services; 28-30 October 1991.

[Shann 49] Shannon, C.E., "The Synthesis of Two-Terminal Switching Circuits," *Bell System Technical Journal, 28:* 59-98 (1949).

[Spick 83] Spickelmier, R. L. and A. R. Newton, "WOMBAT: A New Netlist Comparison Program," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 170-171. New York: IEEE Press; 1983.

[Spick 85] Spickelmier, R. L. and A. R. Newton, "Connectivity Verification Using a Rule-Based Approach," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 190-192. New York: IEEE Press; 1985.

[Stana 77] Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science.* Englewood Cliffs: Prentice-Hall, 1977.

[Sterl 86] Sterling, Leon, and Ehud Shapiro. *The Art of Prolog.* Cambridge: The MIT Press, 1986.

[Takas 88] Takashima, M., et. al., "A Circuit Comparison System with Rule-Based Functional Isomorphism Checking," *Proceedings of the IEEE/ACM Design Automation Conference.* 513-516. New York: IEEE Press; 1988.

[Terma 80] Terman, Chris. Computer Program. "ESIM." 1980.

[Terma 86] Terman, Chris. "Esim," Berkeley Distribution of of Design Tools. Computer Science Division, EECS Department, University of California at Berkeley, 1986.

[Tsui 87] Tsui, Frank F. *LSI/VLSI Testability Design.* New York: McGraw-Hill Book Company, 1987.

[Ullma 84] Ullman, Jeffrey D. *Computational Aspects of VLSI.* Rockville, MD: Computer Science Press, 1984.

[Weste 85] Weste, N. H. and Kamran Eshraghian. *Principles of CMOS VLSI Design A Systems Perspective.* New York: Addison-Wesley Publishing Company, 1985.

[Wing 89] Wing, Jeannette M. "What is a Formal Method," CMU Technical Report, CMU-CS-89-200, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15217, (10 November 1989).

[Wing 90a] Wing, Jeannette M. "A Specifier's Introduction to Formal Methods," CMU Technical Report, CMU-CS-90-13b, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15217, (21 May 1990).

[Wing 90b] Wing, Jeannette M. "A Specifier's Introduction to Formal Methods," *Computer, 23:9* 8-24 (September 1990).

[Yaros 89] Yarost, S. A. *A Circuit Extraction System and Graphical Display for VLSI Design.* MS thesis, AFIT/GCE/ENG/89D-9. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH December 1989 (AD-A215668).

Vita

Captain Michael A. Dukes was born in Lawton, Oklahoma on 3 June 1960. Following graduation from high school at Springfield, Virginia in 1978, he received an appointment to the US Military Academy at West Point, New York. He graduated from the Military Academy in May 1982, with a degree of Bachelor of Science and a commission in the US Army. After completion of the Signal Officer Basic Course, Captain Dukes was assigned to Fort Gordon as a Teleprocessing Operations and Computer Automation Officer within the Directorate of Information Management. Prior to entering the Air Force Institute of Technology, he attended the Signal Officer Advanced Course. Captain Dukes graduated from the Air Force Institute of Technology in September 1988 with a degree of Master of Science in Electrical Engineering. Following graduation, Captain Dukes will be assigned to the Electronic Technology and Devices Laboratory, US Army Laboratory Command, at Fort Monmouth, New Jersey.

Permanent address: 18901 N.E. 237th Terrace
Fort McCoy, Florida 32637

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Doctoral Dissertation

4. TITLE AND SUBTITLE

HARDWARE-VERIFICATION THROUGH LOGIC EXTRACTION

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael A. Dukes, Captain, USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DS/ENG/92-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A Prolog-based system is described which employs logic-extraction to perform hardware-verification. The extraction rules are built automatically from hierarchical structural VHDL models, enabling the equivalence of a structural VHDL description and a layout specification to be verified. Pin-to-pin critical-path analysis is performed within the logic-extraction process; many non-critical paths are pruned early, making pin-to-pin critical path analysis of large circuits feasible. It is demonstrated that a design methodology based on logic extraction, VHDL, and a layout tool can provide a fabricated functionally-correct IC design without circuit-level or switch-level simulation. This methodology is shown to be practical for VLSI designs up to 250,000 transistors in size. The properties of correctness, completeness, and guaranteed termination are examined for the extraction process.

14. SUBJECT TERMS	15. NUMBER OF PAGES
VLSI, VHDL, Prolog, Circuit Extraction, Reverse Engineering, Computer Aided Design, Integrated Circuits, Artificial Intelligence	165
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL